VAST/77to90
Fortran 77 to Fortran 90 Translator

User's Guide
Version 3.1

Pacific-Sierra Research

Document revision record

Document number V7790C.

Edition	Date	Description
1.0	1/92	Initial release.
1.1	3/92	Minor corrections, INCLUDEs to MODULEs.
1.2	7/92	Added switches, more invocation examples.
2.0	1/93	Derived types to/from VAX-compatible structures, documentation of error detection and trace tools, Changed organization of document.
2.1	7/93	Minor improvements and corrections.
2.2	3/94	Minor improvements and corrections.
3.0	4/95	Major revision. Change document organization.
3.1	10/95	Minor corrections and additions.

Preface

This is a guide to the use of VAST/77to90. VAST/77to90 is a programming tool that translates Fortran 77 into Fortran 90. This manual is designed to give Fortran programmers an understanding of VAST/77to90's capabilities and effective use.

Fortran 90 is the new Fortran language standard (ISO and ANSI). This User's Guide does not contain language reference information; for more information about Fortran 90, please refer to one of the many books that have been published on this subject.

An excellent source of information on Fortran 90 is the *Fortran 90 Handbook* (McGraw-Hill, 1992) by Adams, Brainerd, Martin, Smith, and Wagener. This book gives more background and examples than the standard, and still documents the complete language.

Special notices

Table of Contents

1. Introduction

VAST/77to90 translates Fortran 77 into Fortran 90 as described below. In addition, VAST/77to90 has various supplementary features that assist in maintaining Fortran programs, which are described in a later section.

Fortran 77 to Fortran 90

If you have an existing Fortran program and would like it to take advantage of the new features available in Fortran 90, VAST/77to90 will translate it into Fortran 90 for you. This allows you to take old programs and create new, clean efficient versions in Fortran 90 automatically.

VAST/77to90 contains many output formatting features which improve the appearance of the output code. You can specify exactly how you want the output code to look by using some of the many optional parameters, or you can rely on the default standards.

For detailed information on converting Fortran 77 to Fortran 90, see Section 5. In this mode, VAST/77to90 converts DO and IF loops into Fortran 90 array syntax. This conversion process is described in Section 6. For some massively parallel systems, Fortran 90 array syntax will run much faster than the equivalent DO loop. For these systems, VAST/77to90 can be used as an optimizer to get more efficient code from existing programs.

Translating to Fortran 90 can be done with the `v77to90` command. If the input file name ends in `.f`, the translated Fortran 90 file will have a `V` prefixed to it and the `.f` will be changed into `.f90`. For example, if your input file (Fortran 77) is `dusty.f`, then it will be converted to Fortran 90 with the command:

```
v77to90 dusty.f
```

The Fortran 90 output will be in file `Vdusty.f90`. (Common blocks that have been converted to modules will be in files like `V`*`common`*`.m`. Generated interfaces will be in files like *`routine`*`.inf`.)

Additional Features

VAST/77to90 includes several additional features that are useful in working with existing Fortran programs. These features are of primary use for Fortran 77 programs, but they can be applied to work with new Fortran 90 programs as well. The features are:

- Static error detection. VAST/77to90 reports on cases where undefined variables are being used, and other potentially unsafe or incorrect

programming practices. This can be done for one routine or interprocedural error detection can be done for a whole program.
- Code formatting. VAST/77to90 will "tidy" existing Fortran programs, adding spaces and indenting. See section 8.
- Trace debugging. VAST/77to90 will insert print statements into routines that trace all relevant events. Useful for debugging in situations where an interactive debugger is insufficient.

Further Information

Section 2 describes in more detail how to invoke VAST/77to90 with the v77to90 command, and you probably will want to look at this section, if nothing else.

The switches and options that can be given to VAST/77to90 are discussed in section 3 (translation), 4 (listing), and 8 (output formatting).

If you are interested in the specifics of Fortran 77 to Fortran 90 translation, you may want to examine sections 2 (invocation), 3 (translation options), 4 (listing options), 8 (code output format options), 5 (f77 to f90), and 6 (loops to array syntax).

If you are interested in tools to help with Fortran 77 programs, look at sections 8 (code formatting) and 7 (additional features).

2. Invoking VAST/77to90

VAST/77to90 Command line

VAST/77to90 is executed by the command:

```
  v77to90 [-o output] [-l listing] [options]
      [-77to77] input1 [...inputn]
```

- **`output`** = compilable output file. The default output file name is the input file name prefixed by **`V`**. If the output is Fortran 90, the extension of the default output file will be **`.f90`**.
- **`listing`** = VAST/77to90 listing file. Unless this parameter is used, no listing is produced.
- **`options`** = VAST/77to90 options and parameters. See Section 3 (translation options), Section 5 (listing options), and Section 8 (source output options).
- **`input1...inputn`** = Fortran 77 source input files.
- **`-77to77`** = don't translate to Fortran 90, but apply output formatting and static analysis to Fortran 77 source.

VAST/77to90 invoked with no arguments prints a short usage summary, including the VAST/77to90 version number.

Examples: 77 to 90

Below are some examples of using the v77to90 command to translate Fortran 77 to Fortran 90.

Example 1:

To run `sub.f` through VAST/77to90, save the VAST/77to90 listing in file `sub.lst`, and save the translated code in `sub.v.f90` :

```
v77to90 -l sub.lst -o sub.v.f90 sub.f
```

Example 2:

To run `sub.f` through VAST/77to90, turning on aggressive array syntax translation and interface file generation (`-xrf`), turning off creation of modules and translation of computed GOTO into CASE (`-yms`), list argument mismatch problems (`-p4`), change indenting amount to five spaces (`-ZINDAL=5`), and put the listing into file `lst`:

```
v77to90 -xrf -yms -p4 -ZINDAL=5 -l lst sub.f
```

Example 3:

To run `sub.f` through VAST/77to90, formatting the output code but remaining in Fortran 77 (output will be in file Vprog3.f):

```
v77to90 -77to77 prog3.f
```

VAST/77to90 switches and options

VAST/77to90 allows the following switches and options on invocation:
Switches

	enable	disable
translation:	**`[-x`** ***`switches`*** **`]`**	**`[-y`** ***`switches`*** **`]`**
listing:	**`[-p`** ***`switches`*** **`]`**	**`[-q`** ***`switches`*** **`]`**
output format:	**`[-r`** ***`switches`*** **`]`**	**`[-n`** ***`switches`*** **`]`**

Options

trace debug:	**`[-t]`**
pass directive:	**`[-D`** ***`directive`*** **`[:`** ***`routine,routine`*** **`...]]`**
command file:	**`[-F`** ***`filename`*** **`]`**
page length:	**`[-Ppage`** ***`nnn`*** **`]`**
output format:	**`[-Z`** ***`option`*** **`=`** ***`value`*** **`]`**

- ***`switches`*** = concatenated string of switches to enable or disable. (For settings, see section 3.)
- ***`routine`*** = name of Fortran subroutine or function.
- ***`nnn`*** = integer constant

More Information

Transformation options (`-x`,`-y`) are described in section 5, listing options (`-p`,`-q`,`-P`) in section 6, and output format options and parameters (`-r`,`-n`,`-Z`) in section 7. The `-D` parameter is discussed in section 8.

Command File

The `-F` parameter allows redirection of command line input from a file. This is useful when many options are specified (e.g. `-D` parameters), or to insure a uniform set of invocation options over many invocations. Each command must appear on a separate line. Furthermore, spaces cannot be used to separate keywords from arguments (this is different from the command line convention.) For example, in the invocation

```
v77to90 -Finput.dir input.f
```

in input.dir,

```
-o output.f
```

will not work. Rather, use:

```
-ooutput.f.
```

Trace Debugging Option

The `-t` parameter enables trace debugging. This disables all translation, and inserts print statements for each "event" in the program (each branch, assignment, subroutine call, if-test, etc.)

File extensions

The table below summarizes the file types that VAST/77to90 may use or create.

Extension	Description
`.f`	Fortran 77 source.
`.inc`	Fortran INCLUDE (Fortran 90 and extended Fortran 77) (usually used to hold common blocks).
`.f90`	Fortran 90 source.
`.m`	Fortran 90 module: used in place of common to hold global data, may also have internal routines that operate on the data.
`.inf`	Fortran 90 interface file: number, intent, type and attributes of the arguments to a routine -- used to check connections between program units.

.TBL1 {V-.inc} {File of common block that corresponds to a created .m:}
.TBL1 { } {used by VAST-90 to check correspondence of variables,}
.TBL1 { } {can be deleted after all commons have been replaced}
.TBL1 { } {with uses of the module.}

3. Options

Switches

Switches specifying global actions can be passed to VAST/77to90 in the processor invocation command (as described in the preceding chapter), or via the SWITCH source directive.

When translating your program from Fortran 77 to Fortran 90, you may want to use switches to control the form of the output source code, as you may adopt this code as your new source.

There are three kinds of switch control:

- Control of translation (`-x` and `-y` invocation parameters).
- Control of listing (`-p` and `-q` invocation parameters).
- Control of output formatting (`-r` and `-n` invocation parameters).

Translation switches are described below. Listing options are described in the next section, and output formatting options are described in the section after that.

SWITCH directive

SWITCH directives may be inserted into the source program. The format is:

```
CVD$ SWITCH[,-x ss][,-y tt]
           [,-p ll][,-q kk]
           [,-r zz][,-n xx]
           [,output_format_parameters]
           [,-Ppage nn]
```

corresponding to the invocation switch parameters.

Transformation switches

The table below shows the switches that affect the transformation of the input program; these generally apply to the Fortran 77 to Fortran 90 translation mode. `-x` enables these switches (on), `-y` disables these switches (off).

Table 3.1 -- Transformation control switches (`-x/-y`)

Switch	Description	Default
a	Free format input.	off
b	Free format output source code.	on
c	Fortran 90 subset; see description	off
d	Translate F77 declarations to F90 declarations.	on
e	Examine EQUIVALENCE stmnts for data dependency.	on
f	Generate INTERFACE files.	off
g	Move DATA information to initialization expressions	on
h	Change INCLUDEs to MODULEs	on
i	Put COMMONs into INCLUDE files.	off
j	VAX-compatible structures converted to derived types	off
k	Treat D in Column 1 as a comment (OFF: D=blank).	on
l	Transform IF...GO TO loops to DO and DO WHILE.	on
m	Make MODULEs from COMMON.	on
n	Skip all executable transformations.	off
o	Reserved.	-
p	Don't ignore potential data dependencies.	on
q	Reserved.	-
r	Aggressively generate array syntax.	off
s	Replace computed GOTO with CASE.	on
t	Replace Block IF with CASE.	on
u	Reserved.	-
v	Translate loops to F90 array syntax.	on
w	Reserved.	-
x	Generate CYCLE and EXIT from GOTOs in loops.	on
y	Reserved.	-
z	Reserved.	-
1	Insert INCLUDEs to interface files if they exist.	off
2	Insert INCLUDEs of interface files for all calls.	off

As an example, `-yel` causes EQUIVALENCE statements not to be examined for data dependency analysis when translating loops into array syntax, and IF loops not to be converted to DO loops.

A few switches duplicate or overlap the functions of directives. For example, the `-yp` switch is equivalent to the `NODEPCHK` directive with file scope (`CVD$F NODEPCHK`).

Notes on transformation switches:

b. Output file is free format. This is the default. If you want fixed format Fortran 90 output, specify `-yb`.

c. Fortran 90 subset. Primarily for compatibility with various SIMD computers, this switch emphasizes translation into array syntax and turns off various other features. Specifically, it turns on agressive array syntax translation (`-xr`), avoids ALLOCATE when declaring array temporaries, turns off declaration regeneration (`-yd`), does not create MODULE s from

COMMON s (`-ym`), and generates fixed format output rather than free format output (`-yb`).

`d`. Change declarations to use the Fortran 90 attribute form. See section 8.3.

`e`. Examine EQUIVALENCE statements for data dependency. (Discussed in section 6.)

`f`. Generate INTERFACE files. INTERFACE files allow diagnosis of problems involving argument mismatches. See also switches `1` and `2`.

`g`. Move DATA information to initialization expressions. This transformation (on by default) deletes DATA statements where possible and initializes scalar data directly in the new Fortran 90 declaration.

`h`. Change INCLUDEs into MODULEs. If your program already references INCLUDE files, VAST/77to90 will turn them into MODULE files for you. For example, if many parameters are grouped into an INCLUDE, they will be grouped in the MODULE. See section 5.

`i`. Put COMMONs into INCLUDE files. Conflicting COMMON definitions are noted. This turns off switch m, creation of modules from COMMON. See section 8.5.

`j`. Convert VAX-compatible structures to derived types.

`k`. Treat D in column 1 as a comment character. If this switch is off, a D in column one is treated as a blank. This switch provides compatibility with a debugging feature of some compilers.

`l`. Transform IF loops to DO loops. (Discussed in section 6.)

`m`. Convert COMMONs to MODULEs. This is on by default. VAST/77to90 creates module files and USEs them in place of COMMON blocks in the input program. You cannot use this switch in combination with `-xi` (COMMONs to INCLUDE files). See section 5.

`p`. Don't ignore potential data dependencies. `-yp` is equivalent to the NODEPCHK directive with file scope. (Discussed in section 6.)

`r`. Aggressively generate array syntax. By default, VAST/77to90 is conservative in converting loops into array syntax; it only converts loops that will look reasonably clean in array syntax. However, if you want all possible loops converted to array syntax, then you should specify the `-xr` switch. You might want to do this if your target system was a massively parallel processor, for instance. If you are converting to Fortran 90 to obtain a cleaner source code, then the `-xr` switch is not recommended. See section 6 for more information on translation to array syntax and what is controlled by the `r` switch.

`v`. Transform loops into array syntax. `-yv` is equivalent to the SKIP directive with file scope.

`x`. Change GOTOs that target labels just before the end of loops into CYCLEs, change GOTOs targeting labels just after the end of loops into EXITs.

`1`. Insert INCLUDE statements for interface files for each external call, checking first to see that the interface file already exists for the called routine. This is mutually exclusive of the next switch. See also switch `f`.

`2`. Insert INCLUDE statements for interface files for all external calls, without checking first to see that the interface file already exists. Assume that the INCLUDEd files will be created before the Fortran 90 output itself is compiled. This is mutually exclusive of the previous switch. See also switch `f`.

Switches set via the SWITCH directive may be set only once within a routine; that setting applies to the whole routine.

4. VAST/77to90 listing

You may want to exchange information with VAST/77to90 in order to get the best translation of your program, especially if you are translating from Fortran 77 into Fortran 90. Two paths of communication are available for this purpose: (1) VAST/77to90 informs you of the actions it takes on the program (which loops were optimized, which loops were not translated into array syntax and the reasons for their rejection); (2) you can pass information and commands to VAST/77to90 via directives inserted into the program (see section 5), or via global switches on the VAST/77to90 invocation command (see section 3).

The full VAST/77to90 listing consists of four parts: a listing of the input source, a block of diagnostic messages, a listing of the output transformed source, and summaries of overall statistics. Any part of the listing can be separately enabled or disabled via the listing switches shown in the table below. An example of a full listing is also given later in this section.

Source listing

The input source lines are numbered on the listing. Source lines coming from INCLUDEs are numbered as well. These line numbers are used in the messages, output source listing, and summaries. The codes used for the loop disposition graph (when translating from Fortran 77 into Fortran 90) are listed in Table 6.1:

Table 6.1 -- Loop disposition codes

A	No action requested.
D	Data dependent.
E	Deleted.
N	Not chosen.
T	Translation problem.
V	Translated to array syntax.

The numbers in the column at the right edge of the output source listing correspond to input source line numbers, to help in comparing the transformed source to the original source.

Diagnostic messages

VAST/77to90's diagnostic messages appear in a group at the end of the source listing. Each message includes the line number and (if relevant) a variable name. These messages are VAST/77to90's means of notifying you of what problems it encountered in optimizing the loops in the source program. There are several different types of diagnostics.

Optimization inhibition messages

- *Translation Diagnostic*. Describes a problem or potential problem in translating a loop into array syntax.
- *Data Dependency Conflict*. A real or potential feedback from one loop pass to the next prevents the safe use of array operations.

Informative messages

- *Warning Message*. Some potentially troublesome input has been encountered.
- *Note Message*. Additional information that may be of use.

Error messages

These messages are routed to the standard error file, and are more serious:

- *Syntax Error*. A construct that is not legal in Fortran has been encountered. No translation is done for this program unit.
- *Internal Error*. An internal problem with VAST/77to90 has been detected. No further processing is done for this subprogram; translation is attempted again for the next subprogram in the input file. Please report the error.

You can suppress each of these types of messages independently via the `-q` parameter on the VAST/77to90 command line or on the `SWITCH` directive (see section 5). Section 9 contains a full list of VAST/77to90's diagnostics, with further explanation of the messages, and tips on avoiding certain problems.

Following the listing of the transformed source is the Event Summary, which gives overall counts of errors, diagnostics, and loops transformed.

Example listing

Figure 6.1 shows a VAST/77to90 listing of a subroutine containing several loops which are translated to array syntax and a block IF which becomes a CASE.

```
Pacific-Sierra Research VAST/77to90 V0.05I2  16:19:24   1/29/92   FORTRAN 90

    1.         SUBROUTINE FISH
    2.         COMMON /COM1/ A(99), B(99), C(88), E
    3.         DOUBLE PRECISION A, B
    4.         INTEGER C, E
    5.   c
    6.   c...Test on E
    7.         IF ( E.EQ.1 ) THEN
    8. V-      DO 100 I = 1, N
    9. V- 100  A(I)=B(I)
   10.         ELSE IF ( E.EQ.2 ) THEN
   11. V-      DO 200 I = 1, N
   12. V- 200  A(I)=B(I)**2
   13.         ELSE IF ( E.GE.3 ) THEN
   14. V-      DO 300 I = 1, 99
   15. V- 300  IF (B(I).LT.0) A(I)=0.
   16.         ENDIF
   17.         END

-----------------------------------------------------------------------
```

```
      SUBROUTINE FISH
!..Translated by Pacific-Sierra Research VAST/77to90 0.05I2 16:19:24 1/29/92
!-----------------------------------------------
!   M o d u l e s
!-----------------------------------------------
      USE VCOM1
      IMPLICIT NONE
!-----------------------------------------------
!   L o c a l   V a r i a b l e s
!-----------------------------------------------
      INTEGER I, N
!-----------------------------------------------
!
!...Test on E
      SELECT CASE (E)
      CASE (1)
         A(:N) = B(:N)
      CASE (2)
         A(:N) = B(:N)**2
      CASE (3:)
         WHERE (B < 0) A = 0.
      END SELECT
      END SUBROUTINE FISH

--------------------   EVENT SUMMARY FOR ROUTINE FISH    ------------------
WARNING MESSAGES         --     0       SYNTAX ERRORS            --     0
TRANSLATION DIAGNOSTICS   --     0       DATA DEPENDENCY CONFLICTS --     0
LOOPS EXAMINED            --     3       LOOPS TRANSLATED          --     3
```

Figure 6.1 Example VAST/77to90 listing

Listing control switches

Table 6.2 shows the switches that control the format of the listing file. These switches can be used either on the invocation (for example, `-q h`) or on the `SWITCH` directive (e.g., `CVD$ SWITCH,-q h`).

Table 6.2 -- Listing control switches

Switch	Description	Default
a	Reserved.	-
b	List input line nos. in columns 73-80 of output listing.	on
c	List data dependency conflict messages.	on
d	List declarations added by VAST/77to90.	off
e	List event summary at end of routine.	on
f	List fatal error messages.	on
g	List translation diagnostics.	on
h	List input source lines.	on
i	List included lines.	on
j	Reserved.	-
k	Reserved.	-
l	Produce a listing.	off
m	List messages.	on
n	List translated code.	on
o	List optimization notes.	off

p	List loop summary at end of routine.	on
q	Reserved.	-
r	List potential errors.	on
s	List only summary information.	off
t	Terminal listing: format output for 80 columns.	on
u	Show extent and disposition of loops in source.	on
v	List function references.	off
w	List warning messages.	on
x	Reserved.	-
y	List syntax errors.	on
z	List possibly undefined variables.	on

As an example, if you wanted to get a 132-column printer listing with no warning messages and no event summary, you would specify `-q twe`.

Notes on the listing switches:

`b`. List corresponding input line numbers in columns 73-80 of the listing of the transformed source. This switch is valid only if the n switch is on. This listing feature is useful in relating transformed source lines to original source lines.

`d`. List declarations added by VAST/77to90. This switch is valid only if the `n` switch is on.

`i`. List lines that come from INCLUDEd files. When this switch is on, source lines obtained from INCLUDE files are listed. They are identified by a dash following the line number. This switch is valid only if the h switch (list source lines) is on.

`l`. Produce a listing. `-q l` suppresses all parts of the listing (except the initial header, if the t switch is on). `-q l` is equivalent to the NOLIST directive.

`r`. List potential programming errors, as warning messages.

`s`. List only summary information. If the `s` switch is used, none of the other listing switches except t may be used.

`t`. Format the listing for a terminal. `-q t` results in a wide-format listing file, with printer control, pagination, and page headers, suitable for a 132-column line printer.

`u`. Show extent and disposition of loops in the input source listing. "Draws" a bracket alongside each loop, indicating how VAST/77to90 treated the loop (translated to array syntax, too short, data dependent, and so forth).

Listing page length

If you want a paginated, wide format listing suitable for a line printer, use the -q t option. The page length defaults to 66 lines. To change the number of lines per page, use the `-Ppage` parameter. For example,

```
-Ppage 60
```

changes page length to 60 lines per page.

5. Fortran 77 to Fortran 90

This section descibes features of VAST/77to90's Fortran 77 to Fortran 90 translation mode. Translation of loops into array syntax is a special case that is discussed in detail in Section 9.

Fixed format to free format

By default, VAST/77to90 creates free format Fortran 90 source code. This format does not rely on column placement. It uses & at the end of a line to indicate that a continuation is coming, and it uses `!` as the comment character (which can appear anywhere in a line to signify that the rest of the line is a comment.)

You can request fixed format output with the `-yb` switch.

Declarations

VAST/77to90 will change all the declarations into Fortran 90 form by default. Declarations are collected by type and arrays are grouped by identical declarations of shape. Initialization values are sometimes folded into the declarations (for local parameters and some items in DATA statements).

VAST/77to90 generates a separate module file (`vast_kind_param.m`) that contains definitions of data type kinds. This module is then USEd in the translated source to supply definitions of type kinds. This file should be compiled before or with the translated source file.

VAST/77to90 declares all variables appearing in the routine explicitly, and the IMPLICIT NONE statement is added at the top of the routine.

If you prefer to have declarations remain in Fortran 77 form, use the `-yd` switch.

Extensions

Many Fortran 77 extensions are accepted. Some are simply passed through to the output source; others, such as the “byte”-style declaration syntax, are translated to Fortran 90 constructs where possible. One Fortran 77 extension not accepted is the pointer data type. As described later in this section, VAX-style structures can be translated into Fortran 90 derived types.

Example:

```
      real*8 x(100)
```

Translation:
```
      use vast_kind_params
      ...
      real(double) , dimension(100) :: x
```

Control statements

VAST/77to90 changes many older control statements into the newer control statements available in Fortran 90. This reduces the number of GOTO statements and the number of labels in a program, making the flow of control much easier to follow and maintain.

Arithmetic IF to GOTO

Arithmetic IF statements are transformed into IF (...) GO TO statements or if possible block IFs.

Example:
```
      IF ( Y ) 200, 50, 60
 50   CONTINUE
      X = 0.
      GO TO 100
 60   CONTINUE
      X = 1.
 100  CONTINUE
      Y = Y + Z
 200  CONTINUE
```

Translation:
```
      IF (Y >= 0.) THEN
         IF (Y <= 0.) THEN
            X = 0.
         ELSE
            X = 1.
         ENDIF
         Y = Y + Z
      ENDIF
```

GO TO to block IF

VAST/77to90 changes GOTOs into block IFs when possible.

Example:
```
      IF ( X .GT. Y ) GO TO 10
      S = SQRT(T)
      GO TO 20
   10 CONTINUE
      S = EXP(T)
   20 CONTINUE
```

Translation:
```
      IF ( X <= Y ) THEN
         S = SQRT(T)
      ELSE
         S = EXP(T)
      ENDIF
```

The loop below is not converted to block form, as the branches are not properly nested.

Example:
```
      IF ( X .GT. Y ) GO TO 10          (Not converted.)
      S = SQRT(T)
      IF ( Y .GT. Z ) GO TO 20
   10 CONTINUE
      U = EXP(V)
      W = Z**2
   20 CONTINUE
```

Labeled DO to DO/ENDDO

Example:
```
      DO 250 I = 1, n
         read *, x(i), y(i), z(i)
  250 continue
```

Translation:
```
      do I = 1, n
         read *, x(i), y(i), z(i)
      end do
```

GOTO to CYCLE and EXIT

VAST/77to90 translates branches to the end of a loop into CYCLEs, and branches to the statement immediately following the end of a loop into EXITs. Extraneous labels are deleted. CYCLEs and EXITs for non-inner loops are handled; if necessary a new loop construct name is created for the target loop.

Example:
```
      DO 200 J = 1, N
      DO 100 I = 1, N
         IF ( B(I,J).LT.0. ) GO TO 100
         IF ( A(I,J).EQ.0. ) GO TO 300
         B(I,J) = 2.
  100 CONTINUE
  200 CONTINUE
  300 CONTINUE
```

Translation:
```
      L200: DO J = 1, N
         DO I = 1, N
            IF (B(I,J) < 0.) CYCLE
            IF (A(I,J) == 0.) EXIT  L200
            B(I,J) = 2.
         END DO
      END DO L200
```

This transformation can be disabled with the -yx switch; it is on by default.

Computed GOTO to CASE

VAST/77to90 generates CASE statements in place of computed GOTOs where possible. To be eligible for this transformation, computed GOTOs must have a flow of control such that each label of the computed GOTO is accessed only through the computed GOTO statement itself.

Example:
```
      GO TO ( 50, 50, 100 ) I
      B(I) = 2.
      GO TO 200
 50   CONTINUE
      X = 1.
      GO TO 200
 100  CONTINUE
      X = 2.
 200  CONTINUE
```

Translation:
```
      SELECT CASE (I)
      CASE DEFAULT
         B(I) = 2.
         GO TO 200
      CASE (1:2)
         X = 1.
         GO TO 200
      CASE (3)
         X = 2.
      END SELECT
  200 CONTINUE
```

This transformation can be disabled with the -ys switch.

Block IF to CASE

VAST/77to90 generates CASE statements in place of block IF statements where possible. To be eligible for this transformation, a block IF must have at least three branches. Each piece of the block IF must test on the same integer or character variable or expression. An ELSE statement becomes the default part of the CASE construct.

Example:
```
      IF ( I.EQ.1 ) THEN
         X = 1.
      ELSE IF ( I.EQ.2 ) THEN
         X = 2.
      ELSE IF ( I.GT.6 ) THEN
         X = 3.
      ELSE
         X = 4.
      ENDIF
```

Translation:
```
      SELECT CASE (I)
      CASE (1)
         X = 1.
      CASE (2)
         X = 2.
      CASE (7:)
         X = 3.
      CASE DEFAULT
         X = 4.
      END SELECT
```

The transformation of block IFs to CASE statements can be disabled with the -yt switch.

INCLUDEs to MODULEs

While INCLUDE statements are not a part of standard Fortran 77, many existing Fortran 77 systems accept them. If your program uses INCLUDE statements to group related global data, VAST/77to90 attempts to preserve this grouping as it changes the INCLUDEs into MODULEs.

By default, VAST/77to90 attempts to create one MODULE for one INCLUDE. All COMMON elements and parameters are grouped into a single MODULE whose name is derived from the root name of the INCLUDE file. For example, when VAST/77to90 encounters an INCLUDE file called `include_name.inc`, a module named `Vinclude_name` is generated and its output is stored in `Vinclude_name.m`. On subsequent uses of `include_name.inc` VAST/77to90 does not check to see if the INCLUDE file has changed since its first translation. Therefore, whenever an INCLUDE file has been changed, the corresponding `Vinclude_name.m` file should be deleted.

Example:
`i1.inc`:
```
      PARAMETER ( N = 100 )
      REAL A(N), B(N)
      COMMON /C1/ A
      COMMON /C2/ B
```

`one.f`:
```
      SUBROUTINE ONE
      INCLUDE 'i1.inc'
      DO I = 1, N
         A(I) = B(I)
      END DO
      END
```

Translation:
`Vi1.m`:
```
      MODULE VI1
         INTEGER, PARAMETER :: N = 100
         REAL, DIMENSION(N) :: A, B
      END MODULE VI1
```

```
Vone.f90:
      SUBROUTINE ONE
      USE VI1
      A = B
      END SUBROUTINE ONE
```

If you would rather leave the INCLUDEs and not have MODULEs, you can turn off all generation of MODULEs with the -ym switch. If you want MODULEs but do not need them to reflect the INCLUDE grouping, use the -yh switch.

COMMON

VAST/77to90 can change COMMON blocks into module files, or cut COMMON blocks out into INCLUDE files, or leave existing COMMON blocks alone, at your option.

COMMONs to MODULEs

By default, VAST/77to90 changes COMMON blocks into MODULEs. This can be disabled with the `-ym` switch. This will also disable the generation of MODULEs from INCLUDEs. If a COMMON block originates from an INCLUDE statement, then it will be put into a MODULE as described in the previous section; otherwise a new MODULE will be created for the COMMON.

The first time a COMMON is processed, two files are created: *`common_name`*`.m` and *`common_name`*`.vinc`. These contain the MODULE and COMMON information, respectively.

Any subsequent processing of the same COMMON block is then compared to the *`common_name`*`.vinc` file that was created with the first processing. MODULEs can be created as long as the COMMON block elements match in data type and size. A MODULE cannot be created if the COMMON block being processed has more elements than the first translation of the COMMON block into a MODULE. However, it may have less elements. You should translate the largest instance of the COMMON first, if you have a COMMON block with varying lengths.

It is also your responsibility to ensure that all or none of the occurrences of a COMMON block in all routines are translated to a MODULE; if this is not the case, then you should normalize your COMMON blocks or use the `-ym` switch to disable translation to MODULEs. Obviously, global data for items in MODULEs is not shared with global data in COMMON blocks, so the program must use all one or all the other.

In cases where two occurrences of a COMMON block have different names or a conflict with a local variable, VAST/77to90 uses the renaming and/or ONLY capability of the USE statement when referencing the MODULE.

```
      SUBROUTINE ONE
      COMMON / ACOM / A, B, C
      ...
      END
```

```
SUBROUTINE TWO
COMMON / ACOM / AA, BB
REAL C   ! local c
...
AA = 99
BB = 100
...
END
```

This is the MODULE file created by VAST/77to90 (named `Vacom.m`):

```
MODULE VACOM
   REAL :: A, B, C
END MODULE VACOM
```

This is the COMMON file created by VAST/77to90 to match up other uses of the COMMON block before conversion to USE the MODULE (name `acom.vinc`):

```
COMMON / ACOM / A, B, C
```

And this is the Fortran 90 output file:

```
SUBROUTINE ONE
USE VACOM
...
END SUBROUTINE ONE

SUBROUTINE TWO
USE VACOM, ONLY: AA => A, BB => B
REAL C   ! local c
...
AA = 99
BB = 100
...
END SUBROUTINE TWO
```

When a MODULE cannot be created for a COMMON, a warning message is issued and the COMMON block reference remains in the translated output.

COMMONs to INCLUDEs

If you prefer, rather than MODULEs, VAST/77to90 can just create INCLUDE files for each COMMON block. To request this, use `-xi`. This will turn off the `-xm` switch (module creation); you may request COMMONs to be put into INCLUDEs or changed into MODULEs, but not both.

Interface files

Creating interface files

If you specify the `-xf` switch, VAST/77to90 will create an interface file for each routine it processes. This file will be called `*routine*.inf` where "routine" is the subprogram name (interface files have extension `.inf`). You may wish to run

VAST/77to90 in an initial pass just to create interface files for all the routines, and then run it again to include the interface files in the calling routine.

The creation of interface files allows the Fortran 90 compiler to check for argument mismatches in type, number, or kind. This can diagnose many otherwise hard to find programming errors.. It can also allow increased optimization by the Fortran 90 compiler.

Consider this routine:

```
      SUBROUTINE DEMOINTF (A, B, C, S, N)
      PARAMETER (M=50)
      REAL A(N), B(M), C(100)
      COMMON /BLOCK/ X(M)
C
      DO I = 1, N
         A(I) = X(I)
         S = S + B(I)
      END DO
C
      PRINT *, S
C
      CALL SUB1 (S, A, N)
C
      RETURN
      END
```

The interface file generated with the `-xf` switch for this routine looks like:

```
      INTERFACE
!...Generated by VAST/77to90 1.02B 18:25 1/22/92
      SUBROUTINE DEMOINTF (A, B, C, S, N)
      INTEGER M
      PARAMETER (M = 50)
      REAL, DIMENSION(N), INTENT(OUT) :: A
      REAL, DIMENSION(M), INTENT(IN) :: B
      REAL, DIMENSION(100) :: C
!VAST/77TO90: Dummy argument C is not ref. in this routine.
      REAL, INTENT(INOUT) :: S
      INTEGER, INTENT(IN) :: N
!VAST... /BLOCK/ X(IN)
!VAST... Calls: SUB1
      END SUBROUTINE DEMOINTF
      END INTERFACE
```

Including interface files

If you specify the `-x1` switch, VAST/77to90 will check to see if an interface file exists when a subroutine call or function reference occurs in the input source program. If an interface file does exist, VAST will generate an INCLUDE statement to include the proper `.inf` file in the calling routine.

If you specify the `-x2` switch, VAST/77to90 will assume that all interface files are being created, and will generate an INCLUDE statement for each routine called from the current routine, whether the interface file exists yet or not. To get interface files created for all program units, and included in all calling routines, you should use `-xf2`.

VAX-compatible structures

VAX-compatible structure definitions and structure references can be translated into Fortran 90 derived type definitions and references, if you specify the `-xj` switch.

```
      structure /stats/
         integer rbi(5), hrs(5)
      end structure
c
      structure /team/
         character*10 player
         integer number
         record /stats/ statistics
      end structure
c
      record /team/ roster(9)
c
      do i = 1, 9
        write(6,50) roster(i).player
        if ( roster(i).player == ' ' ) go to 100
        do j = 1, 5
          write(6,60) roster(i).statistics.rbi(j)
          write(6,70) roster(i).statistics.hrs(j)
        end do
 100    continue
      end do
c
50    format(2x, ' player: ',a10)
60    format(2x, ' rbis: ',i3)
70    format(2x, '  hrs: ',i2)
      end
```

Translation:

```
      type stats
         sequence
         integer rbi(5)
         integer hrs(5)
      end type stats
!
      type team
         sequence
         character*10 player
         integer number
         type (stats) statistics
      end type team
!
      type (team) roster(9)
!
      do i = 1, 9
        write(6,50) roster(i) % player
        if ( roster(i) % player == ' ' ) cycle
           do j = 1, 5
             write(6,60) roster(i)%statistics%rbi(j)
             write(6,70) roster(i)%statistics%hrs(j)
```

```
            end do
         endif
       end do iloop
c
50     format(2x, ' player: ',a10)
60     format(2x, ' rbis: ',i3)
70     format(2x, '  hrs: ',i2)
       end
```

VAST/77to90 also translates STRUCTUREs that contain UNION and MAP statements. UNIONs allow several different kinds of data items to share the same space in a STRUCTURE.

User directives

You may sometimes have information about the structure of a program and about its data that is unavailable to VAST/77to90 through inspection of individual program units. For this reason, a way for you to guide VAST/77to90 is supplied via *user directives*. User directives are treated as comments by Fortran compilers, thus preserving code transportability.

Most of these directives relate to the translation of loops into array syntax, and are only of interest if you are translating from Fortran 77 into Fortran 90.

VAST/77to90 user directives have the format:

```
    F
    R
CVD$L directive
    b             (this is a blank)
```

The C in column 1 makes the directive a comment for Fortran 77 compilers. (Use an ! in column one instead for free format.) The VD$ flags this line as a directive to VAST/77to90. Following the $ is an optional scope parameter. F stands for "file" (meaning the directive applies until the end of all input files), R stands for "routine" (directive applies until the end of the current routine), and L for "loop" (directive applies to the next loop encountered). A blank following the $ is equivalent to L. Some directives ignore the scope parameter. Directives affecting IF loops must have R or F scope; directives with L scope apply only to DO loops. The body of the directive begins after one or more blanks. Many directives can be preceded by NO, thus effecting the reverse operation.

Examples:

CVD$ NODEPCHK	(Ignore potential data dependencies in the next loop.)
CVD$R SKIP	(Turn off translation into array syntax for the rest of this routine.)
CVD$F LIST	(Turn on listing for rest of file(s).)

The full set of directives is summarized in Table 8.1. The "scope" entry is either `I` for "immediate," meaning that the directive applies immediately; L, meaning that it applies to the next loop; `R`, meaning that it applies to the whole routine; or `LRF`, which means that any of the loop, routine, or file options can be used to control the scope. A short description of each of these directives follows the table. In addition, the more important directives are discussed in detail at the appropriate points in the sections on optimization.

Table 8.1 -- VAST/77to90 directives

Directive	Function	Default	Scope
SKIP NOSKIP	Disable/enable translation.	NOSKIP	LRF
SWITCH	Pass new global switches.	n/a	I
NODEPCHK/ DEPCHK	Do/don't ignore potential data dependencies.	DEPCHK	LRF
NOEQVCHK/ EQVCHK	Don't/do check EQUIVALENCEs to see if they cause data dependencies.	EQVCHK	LRF
PERMUTATION	Pass list of integer arrays that have no repeated values.	n/a	R
NOLIST/ LIST	Turn off/on listing.	LIST	I

SKIP/NOSKIP

SKIP disables translation of loops into Fortran 90 array syntax. NOSKIP only toggles back from SKIP; it does not force translation. The `-yv` switch is equivalent to SKIP with file scope. SKIP is discussed further in section 6.3.

NODEPCHK/DEPCHK

When elements of an array are modified within a loop, VAST/77to90 must determine the exact storage relationship of these elements to all other references to the array in the loop. This must be done to ensure that the references do not overlap, and thus can safely be translated to array syntax. When the relationships cannot be determined, VAST issues a *potential dependency diagnostic*. The NODEPCHK directive asserts that all such potentially recursive relationships are in fact not recursive. It does not, however, force the optimization of operations that are unambiguously directive is used only to toggle back to the default state. The `-yp` switch is equivalent to NODEPCHK with file scope.

NOEQVCHK/EQVCHK

NOEQVCHK directs VAST/77to90 to ignore relationships between variables caused by EQUIVALENCE statements, when examining the data dependencies in a loop. The `-ye` switch is equivalent to NOEQVCHK with file scope.

PERMUTATION

The PERMUTATION directive declares an integer array to have no repeated values. This is useful when the integer array is used as a subscript for another array (indirect addressing). If it is known that the integer array is used merely to permute the elements of the subscripted array, then it can often be determined that no feedback exists with that array reference.

NOLIST/LIST

Listing of the input source can be selectively suppressed with the `NOLIST/LIST` directive pair. If `NOLIST` (or the `-q l` switch) is in force when the `END` statement is encountered, the rest of the listing (messages, translated source, summaries) is suppressed unless specifically enabled via `-p` switches. `SWITCH` directive Set (or change) global switches. Described in section 5.1.1.

Specifying directives on invocation

The `-D` invocation parameter can be used to specify directives without inserting them in the actual input source code. The format is:

```
-D directive[:routine,routine,...]
```

Where *`directive`* is any VAST/77to90 directive, and *`routine`* is a routine in the input source to which the directive is to be applied. If no routine names are supplied, the directive applies to the entire input source. Multiple `-D` parameters can be supplied. A `-F` command line option can be used to redirect command line processing to a file of `-D` commands.

Alternate return

Example:
```
      CALL DOWORK (A, B, N, *800)
      ...
 800  CALL ERROR (5)
```

Translation:
```
      CALL DOWORK (A, B, N, J1)
      IF (J1 .EQ. 1) GO TO 800
      ...
```

Statement functions

Statement functions, are expected to be designated as obsolescent in Fortran 95; they are transformed into internal procedures by VAST/77to90.

Example:
```
      subroutine dowork (s)
      xfunc(a) = a + 1.0/a
      print *, xfunc(s)
      ...
```

Translation:
```
      subroutine dowork (s)
      ...
      print *, xfunc(s)
      ...
      contains
         real function xfunc(a)
         real a
         xfunc = a + 1.0/a
         return
         end function xfunc
```

Other obsolescent features

Other constructs that have been designated “obsolescent” in Fortran 90 but are not transformed by VAST/77to90 include:

- PAUSE statement: replace this with appropriate I/O statements.
- ASSIGN and assigned GOTO: replace this with internal procedures or regular GOTOs or other control structures.
- Hollerith data: replace with character strings. (May be converted by newer versions of VAST/77to90.)

6. Array Syntax

This section outlines general features of VAST/77to90's translation of Fortran 77 loops into Fortran 90 array syntax.

There are two levels of translation for array syntax; the default level, which tries to translate only loops that will be fairly clean after translation, and the aggressive level, which requests maximum translation of loops regardless of the appearance of the output.

If code clarity is your goal, leave the setting at the default. If array syntax has performance implications for your target system (for instance, massively parallel processors) then you may want to use the `-xr` switch to turn on aggressive array syntax translation.

SKIP/NOSKIP directives

You can enable or disable array syntax translaton via the SKIP and NOSKIP directives. By default, translation is on. Translation to array syntax may also be disabled completely by the -yv option switch.

Loop types

Both DO and IF loops are considered for translation to array syntax.

IF loops are converted into DO loops under certain conditions. To be convertible:

- The loop must have a single entrance and a single exit.
- The iteration count for the loop must be determinable at execution time before the loop is entered.
- The IF loop may contain other loops.

The `-yl` switch disables conversion of IF loops to DO loops. All directives (such as NODEPCHK) affecting IF loops must have routine or file scope.

VAST/77to90 analyzes entire nests of DO loops (including converted IF loops) for possible translation to array syntax.

Allowed statements

Statements that may appear in an optimizable loop are listed below.

- Assignment
```
     A(I) = B(I) + C(I)
```

- Conditional assignment
```
     IF ( D(I).LT.0 ) A(I) = B(I) + C(I)
```

- `GO TO label` (label must be forward)
- `GO TO (`*`label1,label2`*`,...)` *`expression`*
  (four or less labels; labels must be forward)
- `IF ( ) GO TO` *`label`* (label must be forward)
- `IF ( ) THEN`
- `ELSEIF ( ) THEN`
- `ELSE`
- `ENDIF`
- Arithmetic IF (all labels must be forward)
  `IF (IJK) 10,20,30`
- `CALL` (handled only upon user direction - either inlined or split out.)
- Comment
- `CONTINUE`
- `FORMAT`
- `DATA`

The appearance of any other statement type in a loop causes that loop to be rejected for translation into array syntax. A diagnostic message points out the offending statement in the listing.

```
      DO 1 I = 1,N        (Not translated.)
      A(I) = B(I)*C(I)
      WRITE (6) A(I),I    (This statement is not translated.)
 1    CONTINUE
```

Statement labels may appear wherever legal.

Loops containing character variables or character constants are not translated to array syntax.

Understanding loop variables

Understanding loop translation is made possible by understanding the translatable and nontranslatable uses of the variables in a loop. Every variable in a translatable loop can be categorized as one of three things: scalar, index, or vector. A scalar is a single value that does not change through all the iterations of the loop. An index is an integer quantity that is incremented by a constant amount each pass through the loop. A vector is a range of memory locations, with a constant skip or stride between consecutive elements. Here is an example of each of these:

Loop:

```
DO 10 I = 1,N
   J = J+1
   A(J) = X
10 CONTINUE
```

Classification:

I,J: index variables
A(J): vector
X: scalar

The following sections examine each of these kinds of variables in detail, and explore how their different uses can allow a loop to translate, or not.

Indexing

This section examines the various ways arrays can be indexed in loops, and describes how they are handled by VAST/77to90. A vector index is an integer variable which is incremented by a constant amount each pass through a loop. A vector index may be defined either in terms of a previously defined vector index or in terms of its own previous value. The loop below demonstrates some types of vector indexes that are handled by VAST/77to90.

Example:

```
      DO 3210 I = 1,50     (Loop index is always a vector index.)
      J = J+1              (Recurrent vector index definition.)
      K = I*2+3            (V.I. defined in terms of another V.I.)
      R(I) = B(K)*C(M)+A(J)
 3210 M = M-4              (Recurrent vector index definition.)
```

Vector indexes in this loop: I, J, K, M

More specifically, a statement setting up a vector index must be transformable into one of these two forms:

Form 1: (vector index1) = (vector index1) + or - (invariant expression)

Form 2: (vector index2) = (vector index1) * (invariant expression)
+ or - (invariant expression)

where an invariant expression is an expression whose value is constant for all passes of the DO loop.

The loop below contains examples of some statements that cannot be changed into one of the forms just presented and thus do not define vector indexes.

Example:
```
      DO 3218 I = 1,N
      J = J*2             (Not a vector index.)
      K = I/4             (Not a vector index.)
      M = I*I             (Not a vector index.)
 3218 A(J) = B(K)*C(M)
```

For an array to be directly accessed as a vector, it must have at least one subscript which is transformable into the form:

(vector index) * (invariant expression) + or - (invariant expression)

Example:
```
      DO 3240 I = 1,N                    (Optimized)
 3240 A(5*I) = B(I-6)+C((I-1)*N+M)
```

Translation:
```
      A(5:5*N:5) = B(-5:N-6) + C(M:N*(N-1)+M):N)
```

Where necessary, VAST/77to90 sets final values of vector indexes to ensure that all variables modified in the original loop are given the same values in the translated code. Do not put index variables in COMMON, or they may be unecessarily saved even when the value may not be needed.

Indirect Addressing

When an array's subscript is itself an array reference with a vector index, the array is said to be *indirectly addressed* or to have a *vector subscript*. The array syntax version of this construct looks very similar to the scalar version.

Example:
```
      DO 3260 I = 1,N
 3260 A(IA(I)) = B(IB(I)*6-IABS(IC(I)))
```

Translation:
```
      A(IA(:N)) = B(IB(:N)*6-IABS(IC(:N)))
```

Such array references will be translated automatically only if they obey these rules:

- An array which is indirectly stored into (scattered) may not appear elsewhere in the loop.
- The vector subscript of this array must have no repeated elements. (This information can be passed to VAST/77to90 with the PERMUTATION directive.)
- An array which is indirectly read from (gathered) may not be stored into within the loop.

In the example below, the appearances of B violate the first rule.
```
DO 3265 I = 1,N
         B(IB(I)) = B(IB(I)) + A(I)
 3265 CONTINUE
```

These rules may be overridden with the NODEPCHK user directive (and in some cases, the PERMUTATION directive), if the user knows that the indexing is non-recursive.

The use of a vector index outside of array subscripts is translated by VAST/77to90 into a reference to an array constructor, which creates an array of values spanning the range of the index.

Example:
```
      DO 3270 I = 1,N
 3270 A(I) = COS(B(I)*I)
```

Translation:
```
      A(:N) = COS(B(:N) * (/ (I,I=1,N) /)
```

Storing into scalars

Scalar variables are unchanging single locations in memory, such as a simple variable (X). Array references whose subscript values are invariant in a loop (and thus represent a single location through all passes of the loop) are called *invariant array references* or *array constants*. Invariant array references are treated similarly to simple scalar variables by VAST/77to90.

Scalar variables that are modified in a loop can sometimes inhibit translation. Scalar variables which are not modified in the loop do not inhibit translation. This section discusses the transformations used to deal with the storing of values into non-index scalars within a loop.

Scalar promotion

When a scalar is set to an array-valued expression, that scalar must be *promoted* to an array. This requires the introduction of temporary arrays which replace the promoted scalars. (This is done only when the aggressive array syntax switch is enabled.)

VAST/77to90 uses static local arrays for temporaries when the size is known at compile time; otherwise it uses allocatable or automatic arrays.

Example:
```
      SUBROUTINE SOLVE (A, B, N)
      ...
      DO 3300 I = 1, N
         S = 1./A(I)                    (S must be promoted.)
         B(I) = SQRT(S) + S
 3320 CONTINUE
```

Translation:
```
      REAL, ALLOCATABLE :: S1U(:)
      ...
      ALLOCATE (S1U(N))
      S1U = 1./A(:N)
      B(:N) = SQRT(S1U) + S1U
      DEALLOCATE (S1U)
```

VAST/77to90 saves the last value of a promoted scalar (the value the scalar would have had on exiting the original loop) only when it determines that the value may be needed following execution of the loop.

Scalar folding

In certain cases, VAST/77to90 will eliminate a promoted scalar by forward substitution. VAST/77to90 folds a scalar only when it eliminates an array temporary and does not add operations. (I.e. the scalar is set to a single array, as in the example below, or the scalar is used only once.)

Example:
```
      DO 3330 I = 1, N
         T = A(I)
         B(I) = T + 1./T
 3320 CONTINUE
```

Translation:
```
      B(:N) = A(:N) + 1./A(:N)
```
(T has been eliminated.)

Carry-Around Scalars

Scalars which may be used before they are defined in a loop are called *carry-around scalars*. They are usually recursive. All references to these variables are collected in a scalar loop and split out from the rest of the calculation if possible.

Example:
```
      DO 3313 I = 1,N            (Not translated.)
      A(I) = S + 1/S             (S is "carried around".)
      B(I) = C(I) - A(I) + S
 3313 S = B(I) + D(I)
```

Reduction Functions

A *reduction function* is an operation which condenses array operands into one scalar value which characterizes some aspect of the input arrays.

The reduction functions translated by VAST/77to90 are:

Operation type	Translated to
`S = S+A(I)`	`SUM`
`S = S*A(I)`	`PRODUCT`
`S = AMAX1(S,A(I))`	`MAXVAL`
`S = AMIN1(S,A(I))`	`MINVAL`
`S = S+A(I)*B(I)`	`DOT_PRODUCT`

`index of max.element`	`MAXLOC`
`index of min. element`	`MINLOC`
`IF (L(I)) N=N+1`	`COUNT`
`L = L .OR. LA(I)`	`ANY`
`L = L .AND. LA(I)`	`ALL`

These operations are recognized in general forms.

Example:
```
      DO 10 I = 1, N
  10  S = A(I) + SQRT(B(I)) + S - (C(I)*D(I))
```

Translation:
```
      S = S + SUM(A(:N)+SQRT(B(:N))-C(:N)*D(:N))
```

VAST/77to90 traces the program flow to find and eliminate initializations for sum and dot product reductions, where possible.

Example:
```
      TOT = 0.0E0
      DO 110 J = M,N
         TOT = TOT + VAL(J)
 110  CONTINUE
```

Translation:
```
      TOT = SUM(VAL(M:N))
```

VAST/77to90 handles conditional reduction functions in loops by using the MASK keyword. COUNT, ANY, and ALL are conditional by definition.

Example:
```
      DO 10 I = 1,N
      IF ( A(I) .GT. 0 ) S = S + B(I)
      IF ( B(I) .GE. D(I) ) X = AMIN1(X,E(I))
      IF ( A(I) .GT. B(I) ) K = K + 1
 10   CONTINUE
```

Translation:
```
      S = S + SUM(B(:N), MASK=A(:N).GT.0)
      X = AMIN1(X,MINVAL(E(:N), MASK=B(:N).GE.D(:N)))
      K = K + COUNT(A(:N).GT.B(:N))
```

VAST/77to90 transforms loops that look for the index of the minimum or maximum element in an array into calls to appropriate Fortran 90 intrinsics (MAXLOC, MINLOC).

Example:
```
      DO 100 I = 1, N
         IF ( A(I).GT.AMX ) THEN
            AMX = A(I)
            IMAX = I
         ENDIF
 100  CONTINUE
```

Translation:
```
      INTEGER I1X(1)
      ...
      I1X = MAXLOC(A)
      IF ( A(I1X(1)) .GT. AMX ) THEN
         AMX = A(I1X(1))
         IMAX = I1X(1)
      ENDIF
```

Conditional operations

This section describes the varieties of conditional statements in loops and how VAST/77to90 deals with them.

VAST/77to90 can translate to array syntax loops containing any combination of conditional assignments, conditional and unconditional forward branching (including arithmetic IFs and computed GOTOs with four or less labels), and block IFs. Because of compilation speed and size restrictions, there is a limit of six simultaneously active conditions. Forward transfers do not have to be properly nested; VAST/77to90 converts all forward transfers into structured IF blocks.

The following conditional statements are not translated to array syntax:

- Computed GOTOs with more than four labels.
- ASSIGNed GOTOs.
- Backward transfers.
- Transfers out of the loop.

There are two basic types of conditions, as illustrated in the loops below. The following loop shows a *loop independent* conditional assignment.
Example:
```
      DO 1 I = 1,100
      B(I) = B(I)+1.
    1 IF (N.EQ.1) A(I) = B(I)+2.
```

Translation:
```
      IF (N == 1) THEN
         B = B + 1.
         A = B + 2.
      ELSE
         B = B + 1.
      ENDIF
```

The IF clause (N.EQ.1) does not depend on the loop index at all; it remains the same for all iterations of the loop. Thus, we know whether or not N is equal to 1 before the loop is executed.

The other kind of condition is shown in the loop below.

Example:
```
      DO 2 I = 1,100
    2 IF (C(I).LT.0) D(I) = E(I)+F(I)
```

Translation:
```
     WHERE ( C.LT.0 ) D = E + F
```

It is called *loop dependent* because the IF clause (C(I).LT.0) can be true on some loop iterations and false on others. Sometimes we want to do the calculation (add E(I) to F(I)) and sometimes we do not. Loop dependent conditions are turned into WHERE and ELSEWHERE statements when code is translated into Fortran 90 array syntax.

The %CD field in the loop summary of the VAST/77to90 listing summarizes what percentage of the operations in the translated loop are loop-dependent conditionals. A value close to 100 in this field means that the loop is almost completely conditional, and may run slower than the scalar original if the condition is sparse (rarely true) depending on how the target system handles WHERE statements.

The loop below shows a *loop-index-dependent* conditional assignment. VAST/77to90 eliminates such conditions by adjusting the limits of the array syntax operation.

Example:
```
      DO 1013 I = 1,100
         IF ( I.NE.J ) A(I) = B(I) + C(I)
 1013 CONTINUE
```

Translation:
```
      IF (J-1.GE.0 .AND. J-1.LE.99) THEN
         A(1:J-1) = B(1:J-1) + C(1:J-1)
         A(J+1:100) = B(J+1:100) + C(J+1:100)
      ELSE
         A(1:100) = B(1:100) + C(1:100)
      ENDIF
```

The `IF (J-1.GE.0...)` test determines whether J is in the range of the DO loop index. If so, the operation is performed up to the J-1 element and then from the J+1 element to N. If J is not in the range of the DO loop index, the operation is performed on all elements.

Functions and Subroutines

In Fortran 90, most intrinsic functions can be applied to array arguments as well as scalars. Thus, references to standard intrinsics do not inhibit translation to array syntax.

Example:
```
      DO 3520 I = 1,N
 3520 A(I) = AMAX1(SIN(SQRT(B(I)+C(I))),D(I))
```

Translation:
```
      A(:N) = AMAX1(SIN(SQRT(B(:N)+C(:N))),D(:N))
```

Outer Loops

Within a nest of loops, VAST/77to90 examines all loops for possible translation to array syntax; more than one loop in the same nest may be translated. To be translatable, loops do not need to be tightly nested, and translatable outer loops can have multiple inner loops. When outer loops are being translated, data dependency messages include the label and the index of the loop they refer to, making it easier to determine the exact location of the dependency.

Conformability

Fortran 90 array syntax requires that in any one expression array references with array subscripts agree in number (*rank*) and size (*extent*) of array dimensions; if this is true, they are said to *conform*.

Fortran 90 transformational instrinsics can be used in most cases to make non-conforming array references conform. The primary transformational intrinsics VAST/77to90 uses are SPREAD (for array references which have "missing" dimensions) and TRANSPOSE and RESHAPE (for array references whose vectorized dimensions are not in the required order). These intrinsics are generated only when aggressive array syntax translation is requested (`-xr`).

Example:
```
      DO 3708 J = 1,50
      DO 3708 I = 1,20
 3708 A(I,J) = A(I,J)+B(I)*C(J)    (B lacks J, C lacks I.)
```

Translation (with `-xr`):
```
      A(:20,:50) = A(:20,:50) +
     1   SPREAD(B(1:20),DIM=2,NCOPIES=50)
     2   SPREAD(C(1:50),DIM=1,NCOPIES=20)
```

Example:
```
      DO 3709 J = 1,M
      DO 3709 I = 1,N
 3709 A(I,J) = 2.0*B(J,I)          (Can't have both I,J and J,I.)
```

Translation:
```
      A(:N,:M) = TRANSPOSE(2.0*B(:M,:N))
```

The next example shows some reduction operations on various dimensions. These are handled with the `DIM=` keyword on the reductions themselves.

Example:
```
C
      DO 200 J=1,M
      DO 200 I=1,N
 200  S=S+A(I,J)+B(I,J)
C
      DO 300 J=1,M
      DO 300 I=1,N
 300  S=S+A(I,J)+B(J,I)
C
      DO 400 J=1,M
      DO 400 I=1,N
 400  D(I)=D(I)+A(I,J)
C
      DO 420 J=1,M
      DO 420 I=1,N
 420  D(J)=D(J)+A(I,J)
```

Translation:

```
 !
       S = S + SUM(A(:N,:M)+B(:N,:M))
 !
       S = S + SUM(A(:N,:M)+TRANSPOSE(B(:M,:N)))
 !
       D(:N) = D(:N) + SUM(A(:N,:M), DIM=2)
 !
       D(:M) = D(:M) + SUM(A(:N,:M), DIM=1)
```

A final example shows some transformational functions used to force conformability on the arrays. The array A is RESHAPEd and then SPREAD to match the shape of array B. (This example is translated only if the -xr switch is specified.)

Example:

```
      DO 10 K = 1,N
      DO 10 J = 1,M
      DO 10 I2 = 1,L2
      DO 10 I1 = 1,L1
      B(I1,I2,J,K) = A(K,I2,I1)
 10   CONTINUE
```

Translation:

```
       B(:L1,:L2,:M,:N) = SPREAD(RESHAPE(
      1   SOURCE=A(:N,:L2,:L1),SHAPE=(/L1,L2,N/),
      1   ORDER=(/3,2,1/)),DIM=3,NCOPIES=M)
```

Outer Loop Inhibitors

There are several constructs which can prevent translation of an outer loop. For instance, when the iteration count of an inner loop is not constant for all passes of the outer loop, then the outer loop is not translated (the inner loops are still translated if possible). Below, the iteration count for both the 100 loop and the 150 loop change with each pass of the outer loop. Thus the 200 loop cannot be translated.

```
      DO 200 J = 1,N              (Not Translated.)
         DO 100 I = 1,J           (Translated.)
 100     A(J,I) = A(J,I)*B(J,I)
         DO 150 I = 1,IN(J)       (Translated.)
 150     C(J,I) = 0.
 200  CONTINUE
```

Sometimes recursion along the inner loop also prevents translation of the outer loop. For example, the DO 400 loop below has feedback on the array A along the inner loop, and also cannot safely be translated along the DO 500 loop.

```
      DO 500 J = 1,M             (Not Translated.)
         DO 400 I = 1,N          (Not Translated.)
 400     A(I+1) = A(I) * B(I,J)
 500  CONTINUE
```

When an inner loop is executed conditionally, the outer loop is not translated. In the example below, the DO 700 loop is not translated since the DO 600 loop inside is executed conditionally.

```
      DO 700 I = 1,N             (Not Translated.)
         IF (A(I).GT.0) THEN
            DO 600 J = 1,N       (Translated.)
 600        B(I,J) = A(I)*J
         ENDIF
 700  CONTINUE
```

Finally, loops which do no indexing are not translated. This sometimes occurs when an outer loop is being used for timing purposes. In the example below, no use is made of the index I or any other index defined for the outer loop DO 900.

```
      DO 900 I = 1,N             (Not Translated.)
         DO 800 J = 1,M          (Translated.)
 800     A(J) = B(J) + C(J)
 900  CONTINUE
```

Matrix Multiply Recognition

When the `-xr` switch is enabled (aggressive array syntax generation), VAST/77to90 recognizes most common forms of matrix multiplication and vector-matrix multiplication and converts them to transformational intrinsic function calls (MATMUL, and sometimes TRANSPOSE).

Example:
```
      DO 10 I = 1, N
      DO 10 J = 1, M
      DO 10 K = 1, L
   10 A(I,J) = A(I,J) + B(I,K)*C(K,J)
```

Translation:
```
      A(:N,:M) = A(:N,:M) + MATMUL(B(:N,:L),C(:L,:M))
```

If the result array is initialized within the loop nest, the addition will be suppressed.

Example:
```
      DO 20 I = 1, N
      DO 21 J = 1, M
   21 A(I,J) = 0.0
      DO 20 K = 1, L
      DO 20 J = 1, M
   20 A(I,J) = A(I,J) + B(I,K)*C(K,J)
```

Translation:
```
      A(:N,:M) = MATMUL(B(:N,:L),C(:L,:M))
```

One or both of the operands may also be transposed.

Example:
```
      DO 35 NMAT = 1, 100
      DO 30 I = 1, N
      DO 30 J = 1, M
      A(I,J,NMAT) = 0.
      DO 30 K = 1, L
   30 A(I,J,NMAT) = A(I,J,NMAT) + B(K,I)*C(K,J,NMAT)
   35 CONTINUE
```

Translation:
```
      DO 35 NMAT = 1, 100
         A(:N,:M,NMAT) = MATMUL(TRANSPOSE(B(:L,:N)),
     1      C(:L,:M,NMAT)
   35 CONTINUE
```

VAST/77to90 also converts matrix-vector multiplication into MATMUL intrinsic calls.

Loop Rerolling

A common technique used to aid performance on some computers is to "unroll" loops. This does not look good in Fortran 90 array syntax, and such loops are rerolled by VAST/77to90.

A rerollable loop must have an explicit constant non-unit increment specified on the DO statement. It must contain no data dependencies. An unrolled summation or dot product must add each operand to the reduction scalar in index order. The number of assignments in an unrolled assignment must be the same as the increment on the DO loop, and each assignment must do the next computation in index order. Below are two examples of loops that VAST/77to90 rerolls.

Example:
```
      DO 100 I = 1, 997, 3
         A(I) = B(I) + C(I)
         A(I+1) = B(I+1) + C(I+1)
         A(I+2) = B(I+2) + C(I+2)
  100 CONTINUE
```

Translation:
```
      A(:999) = B(:999) + C(:999)
```

Example:
```
      DO 200 I = 1, N, 5
         S = S + A(I)*B(I)
     1       + A(I+1)*B(I+1) + A(I+2)*B(I+2)
     2       + A(I+3)*B(I+3) + A(I+4)*B(I+4)
  20 CONTINUE
```

Translation:
```
      S = S + DOT_PRODUCT(A(:(N+4)/5)*5,B(:(N+4)/5)*5)
```

Data Dependency Analysis

VAST/77to90 ensures that the code transformed into array syntax gives the same answers as the original. For certain loops, straightforward translation would result (or could result) in incorrect answers. A loop in which results from one loop pass feed back into a future pass of the same loop is said to have a *data dependency conflict* and cannot be completely translated. (Such a loop is also said to be *recursive*" or to contain *recurrences*.) In these cases, VAST/77to90 detects the problem, reports it to the user, and leaves the loop in its original form.

In certain cases, VAST/77to90 can determine that the problem is limited to a subset of the operations in the loop, and if the agressive array syntax switch (`-xr`) is on, it will cut the loop into translated and non-translated sub-loops. The "%DP" field in the loop summary measures how much of the loop is dependent, i.e. left untranslated. If VAST/77to90 determines that more than a certain percentage of a loop is dependent, it will not translate the loop at all. (The exact percentage depends on various other factors.)

VAST/77to90 examines EQUIVALENCE statements to see if they may be masking recursion, and suppresses any potentially unsafe transformations.

Data Dependency Examples

Figure 6.1 demonstrates the concept of data dependency. Four similar loops are displayed. For each loop, the sequences of instructions that would be executed in scalar mode (one at a time) and in array syntax mode (whole arrays at a time) are also shown. Lower case variables (like "a") stand for new values set in the current loop, while upper case variables (such as "A") stand for old values that were set before the loop started.

It is easy to see that the scalar and array syntax sequences for Figure 6.1A are not the same; the array syntax version uses only old values of A, while the scalar version uses new ones. VAST/77to90 detects that this loop is not safe to translate, puts out a data dependency conflict message, and leaves the loop in its original form (the loop is "rejected"). In contrast, the scalar and array sequences for Figure 6.1B are identical; no feedback of results from one loop pass to another is occurring here. VAST/77to90 recognizes that this loop is safe to translate and does so.

The situation is less clear in Figure 6.1C; here the use of the variable "K" in A's subscript makes the proper scalar sequence impossible to determine at compile time. If K is 1, the loop functions like the recursive loop in Figure 6.1A; if K is -1, the loop is safe to translate, as 6.1B was. When it is not possible for VAST/77to90 to tell if a loop is recursive or not, the loop is said to have *ambiguous subscripting*. Often the user knows that a loop (or perhaps all the loops in a routine or program) is not recursive, even though VAST/77to90 cannot tell, as in 6.1C.

In these cases, the user can assume responsibility and direct VAST/77to90 to ignore potential recursions, via the NODEPCHK directive.

Figure 6.1D shows the same loop as in Figure 6.1A, but with a DO increment of 2 instead of (default) 1. VAST/77to90 detects that the loop is now not recursive, because no results feed back into the calculation. These four similar examples point out the sensitivity of data dependency analysis to offset and stride values of arrays which appear on both sides of the equal sign within a loop.

Figure 6.1 Data Dependency Analysis

```
      a: new value of A          A: old value of A
---------------------------6.1A--------------------------------
ORIGINAL LOOP:                   ARRAY SEQUENCE CORRECT?
    DO 71 I = 2,N                No. We are not using updated
 71 A(I+1) = A(I)*B(I)+C(I)      values of A.

SCALAR SEQUENCE:                 ARRAY SEQUENCE:
    a(3) = A(2)*B(2)+C(2)        a(3) = A(2)*B(2)+C(2)
    a(4) = a(3)*B(3)+C(3)        a(4) = A(3)*B(3)+C(3)
    a(5) = a(4)*B(4)+C(4)        a(5) = A(4)*B(4)+C(4)
    a(6) = a(5)*B(5)+C(5)        a(6) = A(5)*B(5)+C(5)
               :                              :
---------------------------6.1B--------------------------------
ORIGINAL LOOP:                   ARRAY SEQUENCE CORRECT?
    DO 72 I = 2,N                Yes. Sequence is identical.
 72 A(I-1) = A(I)*B(I)+C(I)

SCALAR SEQUENCE:                 ARRAY SEQUENCE:
    a(1) = A(2)*B(2)+C(2)        a(1) = A(2)*B(2)+C(2)
    a(2) = A(3)*B(3)+C(3)        a(2) = A(3)*B(3)+C(3)
    a(3) = A(4)*B(4)+C(4)        a(3) = A(4)*B(4)+C(4)
    a(4) = A(5)*B(5)+C(5)        a(4) = A(5)*B(5)+C(5)
               :                              :
---------------------------6.1C--------------------------------
ORIGINAL LOOP:                   ARRAY SEQUENCE CORRECT?
    DO 73 I = 2,N                ? Depends on K. Here, if 0<K<N
 73 A(I+K) = A(I)*B(I)+C(I)      then array is not right.

SCALAR SEQUENCE:                 ARRAY SEQUENCE:
    a(2+K) = A(2)*B(2)+C(2)      a(2+K) = A(2)*B(2)+C(2)
    a(3+K) = ...                 a(3+K) = A(3)*B(3)+C(3)
               :                 a(4+K) = A(4)*B(4)+C(4)
    Can't show more of sequence  a(5+K) = A(5)*B(5)+C(5)
    because we don't know where                :
    A is being changed.                        :
---------------------------6.1D--------------------------------
ORIGINAL LOOP:                   ARRAY SEQUENCE CORRECT?
    DO 74 I = 2,N,2              Yes (Because stride of 2 makes
 74 A(I+1) = A(I)*B(I)+C(I)      operation non-recursive)

SCALAR SEQUENCE:                 ARRAY SEQUENCE:
    a(3) = A(2)*B(2)+C(2)        a(3) = A(2)*B(2)+C(2)
    a(5) = A(4)*B(4)+C(4)        a(5) = A(4)*B(4)+C(4)
    a(7) = A(6)*B(6)+C(6)        a(7) = A(6)*B(6)+C(6)
    a(9) = A(8)*B(8)+C(8)        a(9) = A(8)*B(8)+C(8)
               :                              :
```

Data dependency analysis must extend to more than just single line loops, of course; in the loop shown below, the reference to A at the top of the loop conflicts with the store into A at the bottom. VAST/77to90 prints a message to this effect and does not translate the loop.

```
      DO 1 I = 2,N                  (Not translated.)
      TEMP = A(I-1)+A(I-2)
      B(I) = TEMP+3.0+A(I)
    1 A(I) = SQRT(B(I))-5.0
```

VAST/77to90 uses information from other array dimensions as part of its analysis where possible. The loop in the following example is translated because VAST/77to90 can see that the second dimension indices of the A references can never be equal (since N is not equal to N+1) and thus there is no recursion (The references to A are to two totally different column vectors - they do not share data).

```
      DO 2 I = 2,N                  (translated.)
    2 A(I,N) = A(I-1,N+1)*B(I)+C(I)
```

It is possible for array constants (array references with invariant subscripts) to conflict with "vector" array references. Because of this, the following cannot be safely converted into array syntax operations (J may be between 2 and N):

```
      SUBROUTINE UNSAFE ( A, B, N, J )
      REAL A(*), B(*)
      DO 3 I = 2,N              (Not translated.)
    3 A(I) = A(J)-B(I)
```

Reference Reordering

Sometimes a loop does not contain true feedback but still cannot be translated as-is because of the order in which array elements are referenced. In cases like the example below, if agressive array syntax is requested, VAST/77to90 creates a temporary to hold the "old" elements of an array so that they are still available when needed later. This allows the loop to be safely translated.

Example:
```
      DO 40 I=1,N
      A(I) = B(I)+C(I)+D(I)
      D(I) = E(I)+A(I+1)
 40   CONTINUE
```

Translation:
```
      REAL A1U(N)
      A1U = A(2:N+1)            (Temporary to hold values.)
      A(:N) = B(:N) + C(:N) + D(:N)
      D(:N) = E(:N) + A1U
```

VAST/77to90 also reorders entire statements to eliminate dependencies.

Ambiguous Subscript Resolution

When it is not possible with information contained in the loop to determine the storage relationship between two references to an array, the situation is called *ambiguous subscripting* or *potential feedback*. In these situations, VAST/77to90 looks for statements outside of the loop that may provide information which clears up the ambiguity. In the loop below, VAST/77to90 finds the assignment to N2 which makes it clear that N1 is never equal to N2, and thus there is no feedback.

```
      N2=N1+1
      DO 100 I=1,N          (translated.)
 100  A(I+1,N1) = A(I,N2)*B(I)
```

Loop peeling

When data dependency is caused by one or a few iterations at one end of the DO index range, *loop peeling* can remove the dependency.

Example:

```
      DO 100 I = 1, 30
         A(I) = A(1) + B(I)
 100  CONTINUE
```

Translation:

```
      A(1) = A(1) + B(1)
      A(2:30) = A(1) + B(2:30)
```

If the first iteration is peeled from the loop (executed as a single scalar statement), the rest of the loop iterations can be optimized. Up to three iterations can be peeled from the beginning or end of a loop. The loop may be of variable length, but the stride (increment) must be a constant.

NODEPCHK -- declaring non-recursion

As mentioned previously, the NODEPCHK directive gives the user the ability to direct VAST/77to90 to ignore potential data dependencies in a loop. This capability should be used only when the user knows that no real recursion exists. When it detects potential feedback, VAST/77to90 issues a message asking the user to apply this directive if the loop is in fact not recursive.

Let's look at several examples of situations where the user may want to disable data dependency checking.

In the following loop, VAST/77to90 plays it safe and does not translate, but it is clear to the programmer that K will always be less than or equal to zero, and thus the loop is never recursive:

```
      K = -ABS(M)     (Not translated.)
      DO 1 I = 1,N
  1   A(I+K) = A(I)+B(I)
```

Below, a CVD$ NODEPCHK directive is used to allow the loop to translate.

```
      K = -ABS(M)
CVD$  NODEPCHK          (translated.)
      DO 2 I = 1,N
 2    A(I+K) = A(I)+B(I)
```

In this loop, VAST/77to90 cannot be sure that N1 does not equal N2 and thus rejects the loop:

```
      SUBROUTINE MOVE ( A, B, N, N1, N2 )
      REAL A(N,*), B(*)
      DO 3 I = 1,N      (Not translated.)
  3   A(I+1,N1) = A(I,N2)+B(I)
```

If the user knows that N1 is never equal to N2, then a directive can be inserted as shown here:

```
CVD$  NODEPCHK
      DO 4 I = 1,N       (translated.)
  4   A(I+1,N1) = A(I,N2)+B(I)
```

Finally, the user can see that the array constant A(N*N) in this loop is not in the range of the vector A(I) (I = 1 to N) and thus a directive can be used:

```
CVD$  NODEPCHK            (translated.)
      DO 6 I = 1,N
  6   A(I) = B(I)-A(N*N)
```

NOEQVCHK -- non-recursion in equivalences

It is very rare in real-world programs that recursion is hidden through the use of EQUIVALENCE statements. However, VAST/77to90 must assume the worst and not translate in situations where EQUIVALENCE statements could cause feedback. In programs where many of the variables are EQUIVALENCEd together, this can result in most of the loops being left un-translated.

The NOEQVCHK directive is provided to tell VAST/77to90 that EQUIVALENCE statements can be ignored for data dependency analysis. Use of this directive asserts that variables with different names do not overlap in storage. (This is almost always the case, anyway.) Equivalence checking can be suppressed for the entire input file via the VAST/77to90 command line or the SWITCH directive `-ye`.

In the example below, several local arrays have been equivalenced to a large array in common (perhaps to save space). If the arrays could overlap (for instance, if the value of the variable N was 1500 in the DO 100 loop) then the DO 100 loop cannot be translated. However, as we know the arrays do not overlap, the NOEQVCHK directive is used for the whole routine.

```
      COMMON /BIG/ POOL(100000)
      DIMENSION A(1),B(1),C(1)
      EQUIVALENCE (POOL(1),A(1)),(POOL(1001),B(1)),
     1            (POOL(2001),C(1))
CVD$R NOEQVCHK         (Don't worry about equivalences.)
      .
```

```
          .
          DO 100 I = 1, N       (Translated.)
             A(I+IA) = B(I+IB) + C(I+IC)
     100  CONTINUE
```

PERMUTATION -- declaring safe indirect addressing

When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, feedback is possible even if the subscript is identical. Feedback will occur if there are any repeated elements in the subscript.

Sometimes, indirect addressing is used because the elements of interest in an array are sparsely distributed; in this case an integer array is used to point at the elements that are really wanted, and there are no repeated elements in the integer array (e.g., the example below).

This information can be passed to VAST/77to90 through the PERMUTATION directive. PERMUTATION asserts that the named integer arrays contain no repeated elements (i.e., they serve merely to permute the elements of the arrays they indirectly address).

Syntax:

```
   CVD$ PERMUTATION ( ia1, ia2, ... , ian )
```

PERMUTATION declares the integer arrays (ia1, etc.) to have no repeated values for the entire routine.

Example:

```
   CVD$  PERMUTATION ( IPNT )   (IPNT has no repeated values.)
         ...
         DO 100 I = 1, N
            A(IPNT(I)) = A(IPNT(I)) + B(I)
    100  CONTINUE
```

7. Additional Features

Static analysis of variables

In 77-to-90 or 77-to-77 mode, if a listing is requested (`-l` *`filename`*), VAST/77to90 analyzes all variable uses and definitions, and diagnoses likely or potential programming errors. Conditions checked for include: use of a variable that is currently undefined, unused definition of a variable, and appearance of a variable only once, as a formal argument to an external.

Example:

```
 1.      subroutine test1 ( b, c, d, x )
 2.      if ( b .gt. 0 ) then
 3.         a = 1.0/sqrt(b)
 4.         x = x + a
 5.      else if ( c .gt. 0 ) then
 6.         a = 1.0/sqrt(c)
 7.         x = x + a
 8.      else if ( d .gt. 0 ) then
 9.         x = x + a
10.      endif
11.      end

Line  Typ Message   (W=Warning, N=Note)

 9     W  *** VARIABLE USED BUT NOT DEFINED *** (A)
```

Example:

```
1.      subroutine test2 (a, n)
2.      real a(*)
3.   10 continue
4.      i = i + 1
5.      a(i) = 0
6.      if ( i .lt. n ) go to 10
7.      end

Line  Typ Message   (W=Warning, N=Note)

 4     W  *** VARIABLE USED BUT NOT DEFINED *** (I)
```

Example:

```
1.      subroutine test3
2.      xxx1 = 0.0
3.      call sub1 ( xx1 )
4.      if ( xxx1 .gt. 0 ) call sub2
5.      end
```

```
    Line  Typ Message   (W=Warning, N=Note)

     3     N  Variable appears only as formal argument.
    (XX1)
```

Example:

```
    1.      subroutine test4 ( x, y )
    2.      var1 = 1.0
    3.      if ( x .gt. 0.0 ) varl = sqrt(x)
    4.      y = 1.0/var1 + x
    5.      end

    Line  Typ Message   (W=Warning, N=Note)

     3     W  VARIABLE DEFINED BUT NEVER USED (VARL)
```

Example:

```
    1.      subroutine test5
    2.      common /block/ a, y
    3.      if ( a .gt. 0 ) then
    4.         x = 1.1
    5.      endif
    6.      y = sqrt(x)
    7.      end

    Line  Typ Message   (W=Warning, N=Note)

     6     N  Variable may be used before defined. (X)
```

Trace debugging

In trace debugging mode, VAST/77to90 inserts PRINT statements into the source to trace runtime events, such as assignments, branches, and calls. Trace debugging is enabled via the `-t` parameter. If this option is enabled, no other options are valid. Multiple routines can be traced.

An advantage of this feature is that all events in traced subroutines can be looked at in batch output. This can be very helpful when trying to isolate the source of a problem. A disadvantage is that a large amount of output potentially can be generated. We recommend that this feature be used selectively, only on particular routines of interest. Future implementations will allow directive-level control, so that sections of code with a subprogram can be traced.

Example:

```
      subroutine trace (a, n, s)
      real a(n,n)
      common /block/ i, j, x, y(100)
      a(i,j) = x
      if (y(1) .lt. -3.14) go to 200
      if (y(2) .gt. 0) call sub (x,n)
      do 100 k = 1, n
```

```
         s = s + a(k,1)
 100  continue
 200  continue
      end
```

Translation with `-t`:

```
      subroutine dumbo (a, n, s)
      real a(n,n)
      common /block/ i, j, x, y(100)
       PRINT *, ' DUMBO, Line 4  *  dummy argument  n=', n
       PRINT *, ' DUMBO, Line 4  *  dummy argument  s=', s
      a(i,j) = x
       PRINT *, ' DUMBO, Line 4  *  subscript  i=', i
       PRINT *, ' DUMBO, Line 4  *  subscript  j=', j
       PRINT *, ' DUMBO, Line 4  *  a(i,j)=', a(i,j)
      if (y(1) .lt. (-3.14)) then
       PRINT *, ' DUMBO, Line 5          *  y(2) .gt. 0'
       go to 200
      endif
      if (y(2) .gt. 0) then
          PRINT *, ' DUMBO, Line 6  *  y(2) .gt. 0'
          PRINT *, ' DUMBO, Line 6  *  CALL sub'
         call sub (x, n)
          PRINT *, ' DUMBO, Line 6  *  argument  x= ',x
          PRINT *, ' DUMBO, Line 6  *  argument  n= ',n
      endif
      do k = 1, n
          PRINT *, ' DUMBO, Line 7  *  k=', k
         s = s + a(k,1)
          PRINT *, ' DUMBO, Line 8  *  s=', s
      end do
  200 continue
       PRINT *, ' DUMBO, Line 10  *  LABEL', 200
      end
```

Sample output:

```
   DUMBO, Line 4  *  dummy argument  n=   2
   DUMBO, Line 4  *  dummy argument  s=   0.0
   DUMBO, Line 4  *  subscript  i=   1
   DUMBO, Line 4  *  subscript  j=   2
   DUMBO, Line 4  *  a(i,j)=   3.4
   DUMBO, Line 6  *  y(2) .gt. 0
   DUMBO, Line 6  *  CALL sub
   DUMBO, Line 6  *  argument  x=   2.2
   DUMBO, Line 6  *  argument  n=   2
   DUMBO, Line 7  *  k=   1
   DUMBO, Line 8  *  s=   0.75
   DUMBO, Line 7  *  k=   2
   DUMBO, Line 8  *  s=   2.35
   DUMBO, Line 10  *  LABEL    200
```

8. Output formatting

This section describes the output formatting feature of VAST/77to90. You can select from many options and switches to get your code reformatted in the way you prefer.

Output formatting options fall into two categories: switches (single features that can be turned on or off) and parameters (numerical values). Output formatting switches and parameters may be specified either on the VAST/77to90 invocation line or on the `SWITCH` directive.

Option switches

The switches in the table 8.1 can be specified by using the `-r` and `-n` parameters on the VAST/77to90 invocation line or on the `SWITCH` directive.

Table 8.1 -- Output formatting switches

Switch	Description	Default
a	Place inline comments above statement, if they do not fit	on
b	Put space after a comma in a subscript (e.g. A(X, Y)).	off
c	Put space after a comma in a list (e.g. CALL X (A, B, C)).	on
d	Indent first/last lines of DO loops.	off
e	Put spaces around equal sign (=).	on
f	Move format statements before END statement.	off
g	Put spaces around subscript parentheses.	off
h	Force labels to be on CONTINUE and FORMAT stmts. only	on
i	Indent first/last lines of IF blocks..	off
j	Put spaces around the ** and // operators.	off
k	Put spaces around .AND., .OR., .EQV.and .NEQV. only.	off
l	Convert Fortran output to lower case.	off
m	Squeeze two-line statements into one by ignoring spacing rules.	on
n	Put spaces around non-subscript parentheses.	off
o	Put spaces around all logical operators.	off
p	Put spaces around + and - .	on
q	Spacing around logical operators (see notes).	on
r	Generate comments listing external references.	off
s	Follow spacing rules for operators inside parentheses.	off
t	Put spaces around * and /.	off
u	Add keywords UNIT= and FMT=.	off
v	Put spaces around = in keyword lists.	off
w	Indent first/last lines of WHERE blocks	off
x	`RENUMB=100:10,FORMAT=900:10,TDYON=R`	off
y	Reserved.	--
z	Create output file	on
1	Terminate DO loops with ENDDO.	on
2	Align equals signs for assignment statements in the same block.	off

Notes on output formatting switches:

a. The column in which the inline comment begins is determined by the length of the comment and by the `ILCCOL` parameter described later in this chapter. If the original Fortran line is not reformatted, this switch may still have an effect if the `ILCCOL` parameter is set to a column less than the column in which the original inline comment begins. This switch has no effect on inline comments that remain inline.

c. Lists are any collection of variable and array names, such as those found in the argument lists of SUBROUTINE and FUNCTION statements, CALLs to subroutines or functions, and the input/output list of READ, PRINT, and WRITE statements.

f. This switch causes all of the FORMAT statements to be collected at the end of the module. If this switch is off (the default), the FORMAT statements remain where they were originally. If this switch is on, renumbering will automatically give FORMAT statements the highest labels, because they will now be at the end of the routine. This situation may change if the `FORMAT=` parameter, described later in this section, is used. (Indentation and spacing of FORMAT statements is never changed.)

h. Fortran allows statement numbers on any statement, but the only statement numbers necessary are those used as targets for branches, terminal statements of a DO loop, and FORMAT statements. Branch targets and terminal statements of a DO loop can be converted to CONTINUE statements, and numbered accordingly. With this switch on (the default), this is done automatically.

k. Put spaces around .AND., .OR., .EQV., and .NEQV. only. The default is off. For IF statements, the `k`, `o`, and `q` options are interrelated. Only one may be on at a time. The `o` switch is: put spaces around all logical operators. Default is off. The `q` switch is: put spaces around all logical operators when alone in an IF statement, else around .AND., .OR., .EQV. and .NEQV. only.

Example:
```
LOGICAL L1, L2
IF (A(I).GT.B(I)) A(I) = B(I)
IF (C(I).GT.D(I).AND..NOT.L1) C(I) = D(I)
IF (L1 .AND. .NOT.L2) L2 = .TRUE.
```

This example becomes, with the k switch on and the o and q switches off:
```
LOGICAL L1, L2
IF (A(I).GT.B(I)) A(I) = B(I)
IF (C(I).GT.D(I) .AND. .NOT.L1) C(I) = D(I)
IF (L1 .AND. .NOT.L2) L2 = .TRUE.
```

with the o switch on and the k and q switches off:
```
LOGICAL L1, L2
IF (A(I) .GT. B(I)) A(I) = B(I)
IF (C(I) .GT. D(I) .AND. .NOT. L1) C(I) = D(I)
IF (L1 .AND. .NOT. L2) L2 = .TRUE.
```

and with the q switch on and the k and o switches off:
```
LOGICAL L1, L2
IF (A(I) .GT. B(I)) A(I) = B(I)
IF (C(I).GT.D(I) .AND. .NOT.L1) C(I) = D(I)
IF (L1 .AND. .NOT.L2) L2 = .TRUE.
```

l. This switch causes VAST/77to90 to convert all of the executable part of the code to lowercase. VAST/77to90's declarations will be generated in lowercase as well. The user's declarations will be copied in the same case as they were in the input program. (Character constants and comments are also not converted.)

m. Sometimes, strict adherence to the output formatting spacing rules yields two-line statements that would be more readable as one line. By ignoring some of the spacing rules, the statement is made to fit on one line. If you want this automatic squeezing to be turned off, use `-n m`.

n. Non-subscript parentheses occur in IF, CALL, READ, WRITE, SUBROUTINE, FUNCTION, and other statements.

o. See explanation under switch `k`.

q. See explanation under switch `k`.

r. Generate a block of comments that lists the subroutines and external functions called by the current program unit (including inlined routines). This block of comments is inserted into the translated program unit preceding the first executable statement. Routines are listed in first-occurence order. If no subroutines or external functions are called, a comment to that effect is inserted.

s. This switch uses the settings of the `j`, `p`, and `t` switches to determine spacing inside of parentheses. If the switch is off, as is the default, then no spacing will take place.

u. Fortran allows the keywords UNIT= and FMT= in input and output statements. Most codes, however, use the positional notation. If desired, the keywords will be added and formatted in the same manner as any other keywords in the input or output statement.
Note that if the original statement contains a `UNIT=` or `FMT=` keyword, and the switch is off, the keywords will still remain. This switch does not remove any keywords from the original statement.

x. A shorthand switch for invoking `RENUMB=100:10, FORMAT=900:10, -r R`. Causes renumbering of labels and insertion of comment block summarizing externals.

z. Create an optimized source file. This switch may be turned off if the diagnostic listing only is wanted. Turning this switch off may speed compile time and reduce disk space used. The setting of this switch does not affect the listing of the transformed source in the listing file.

Output format parameters

The following output format parameters can appear on the VAST/77to90 command line or on the SWITCH directive. On the VAST/77to90 command line, these parameters are preceeded by the `-Z` flag (e.g. `-Z RENUM=100:10`)

Label renumbering

`RENUMB=m:n`
This parameter controls the renumbering of the statement labels within a routine. If only the `RENUMB=` part appears without a number following the =, then the

statement labels are renumbered starting at 100 and increasing by 10 for each successive label.

If the parameter is given as `RENUMB=m`, where `m` is any number (less than 99999), then the increment defaults to 10. If the parameter is given as `RENUMB=:n`, then the starting number defaults to 100. The forms `RENUMB=m:` and `RENUMB=:` are not allowed. Use `RENUMB=n` or `RENUMB=`, respectively.

`FORMAT=m:n`

This parameter works exactly like the `RENUMB` parameter, but only on FORMAT statements. If a different numbering scheme is desired for FORMAT statements than for other statements, this parameter is used. If `FORMAT=` is used alone, the default is a starting number of 900 and an increment of 10. The forms `FORMAT=m` and `FORMAT=:n` are also valid as described in the `RENUMB=m:n` section above.

Label alignment

`LABELS=n:l`

This parameter controls placement of statement labels within the statement label field. For the labels to be left-justified, `l` should be set to the character `L`. For the labels to be right-justified, `l` should be set to the character `R`. The column in which the labels are justified is given by `n`, which must be a number from 0 to 5. If `n` is 0, no alignment of statement labels occurs. The default is to right-justify labels in column 5 (`5:R`).

Fortran 90 construct names are treated as part of the statement they apply to, and indented accordingly, with the remainder of the statement following after a single space.

Indentation

There are five indentation parameters:

```
INDDO=n
INDIF=n
INDWH=n
INDCN=n
INDAL=n
```

where `n` is the number of spaces to indent. In each case, `n` must be a number between 0 and 10, inclusive.

INDDO is for DO blocks, INDIF for IF blocks, INDWH for WHERE blocks, and INDCN is the number of spaces to indent continued lines. INDAL sets all the other parameters to n (INDDO, INDIF, INDWH, and INDCN). The default for all of these parameters is 3.

For a DO block, the DO statement and the terminal statement of the DO are placed at the same indentation level, and statements between them occur at the new indentation level. For an IF block, the IF statement itself, any ELSE IF or ELSE statements, and the ENDIF statement all occur at the same indentation level, and statements between them occur at the new indentation level.

Continued lines are indented n spaces from the initial line, where n is given by the INDCN parameter.

```
LSTCOL=n
```

This parameter gives the last column available for indentation, that is, no statement will be started after this column. The default is 31.

Indentation is preserved up to the column limit given by LSTCOL, and no statement begins after column n. Using the defaults gives eight separate block levels, each indented by three spaces from the previous level.

Inline comments

```
ILCCOL=n
```

This parameter aligns inline comments starting at column n. (For fixed-format output, if n is less than 15 or greater than 66, no alignment is done.) Column n is the column in which the inline comment delimiter will appear.

The default is to align inline comments at column 50 (`ILCCOL=50`). To turn off alignment of the inline comments, use `ILCCOL=0`. If an inline comment will fit on the original line, even after reformatting, but will not start at the column used for alignment, it will be placed on the original line so that the last character of the comment will be in the last position of the line. If an inline comment does not fit on the same line as before, a comment will be inserted above the line (or below if the A switch is off), and the comment will be aligned to the appropriate column if possible.

Continuation lines

```
CONCHR=*
```

For fixed-format output, this parameter determines what character to use for continued lines. By default, the successive numbers 1,2,3,...8,9 are used. The tenth continued line uses the decimal point and then the cycle begins over again with 1. If the `CONCHR=` parameter is used, then the character following the = is used as the continuation character. If that character is 0, the default is used, because 0 is not a valid continuation character. (This parameter applies to fixed format source only.) A statement which must be continued on another line may be broken at any point, except within a name or a constant.

Output line length

```
Zolen=n
```

For free format output, this parameter allows specification of the output line length (up to 132 columns).

FORMAT and DATA statements

The output formatting does not change the indentation or spacing inside FORMAT and DATA statements. (However, FORMAT statements may optionally be renumbered and/or moved to the end of the routine.)

Example

Here is an example of VAST/77to90 formatting a Fortran 77 routine. This example shows the use of several switches, but you can rely on the defaults as well.

Here is the input:

```
CVD$  SWITCH,-rfjnostuv2,renumb=10:5
      SUBROUTINE TIDYX1
      REAL A(100),B(100),C(100),D(100),E(100)
      REAL AA(100,100),BB(100,100),
     1   CC(100,100),DD(100,100)
      DO 111J=1, 100
      A(J)=SQRT(B(J))
111   CONTINUE
9999  FORMAT (3F12.8)
      DO 132 J=1,100
      S = B(J)-D(J)*E(J)      ! note alignment
      B(J)=D(J)+E(J)/S**2  ! of inline comments.
      IF (A(J).GT.B(J).AND.C(J).LT.D(J)) GO TO 100
      C(J)=B(J)/C(J)
      GO TO 19
100   CONTINUE
      C(J)=C(J)/B(J)
 19   CONTINUE
      DO 2 I =1,100
      A(J)=AA(I,J)
      AA(I,J)=A(J)+B(I)/BB(I,J)
    2 CC(I,J)=DD(I,J)
      WRITE(6,9999,ERR=154) A(J),B(J),C(J)
132   CONTINUE
  154 RETURN
      END
```

And this is the cleaned-up version:

```
   SUBROUTINE TIDYX1
   REAL A(100),B(100),C(100),D(100),E(100)
   REAL AA(100,100),BB(100,100),
  1   CC(100,100),DD(100,100)
   DO J = 1, 100
      A(J) = SQRT ( B(J) )
   END DO
   DO J = 1, 100
      S    = B(J) - D(J) * E(J)    ! note alignment...
      B(J) = D(J) + E(J) / S ** 2  ! of inline comments.
      IF (A(J) .LE. B(J) .OR. C(J) .GE. D(J)) THEN
         C(J) = B(J) / C(J)
      ELSE
```

```
            C(J) = C(J) / B(J)
         ENDIF
         DO I = 1, 100
            A(J)     = AA(I, J)
            AA(I, J) = A(J) + B(I) / BB(I, J)
            CC(I, J) = DD(I, J)
         END DO
         WRITE ( UNIT=6, FMT=15, ERR=10 ) A(J), B(J), C(J)
      END DO
   10 CONTINUE
      RETURN
   15 FORMAT (3F12.8)
      END
```

9. Diagnostic Messages

This section shows selected VAST/77to90 messages, grouped by category. Most of the messages described here have to do with translation of loops into array syntax.

Data dependency conflicts

Data dependency messages are always followed by the name of the variable which is causing the problem. If outer loop translation is being attempted, the label and index of the loop causing the problem is given as well.

"Feedback of array elements"

```
     DO 4 I=1,N
   4  A(I+1) = A(I) + B(I)
```

Feedback of results makes the loop recursive and thus unsafe to translate to array syntax.

"Feedback of scalar value from one loop pass to another"

```
      DO 112 I = 1,N
      A(I) = A(I) + SCA
  112 SCA = A(I)/C(J)
```

The variable SCA is used in the first line of the DO loop to set A(I), and then is set to a function of A(I) in the last line. This creates feedback of elements of A from one loop pass to the next, which prevents translation to array syntax. SCA is called a "carry-around" scalar, as it carries a value around to the next pass of the loop.

"Potential feedback of array elements -- use directive if ok"

```
      DO 3 I=1,N
    3 A(I+J) = A(I) + B(I)
```

It is not clear whether there is feedback between the two uses of A in this loop or not (it depends on the value of J). Loops of this kind that the user is sure are safe can be translated by putting the directive CVD$ NODEPCHK in front of the loop. Here is another case where this message would result:

```
      DO 1 I=1,N
    1 B(I) = B(IB(I))+A(I)
```

B(IB(I)) is a gathered array, and as the values in IB(I) are unknown, it may conflict with the assignment of elements of B on the left side of the equal sign. If the pattern of B(IB(I)) is known not to overlap B(I), then the NODEPCHK directive should be used.

As a convenience, the d option switch or NODEPCHK directive with routine or global scope may be used instead of loop-by-loop directives.

"Multiple store conflict"

```
      DO 100 I = M,N
      A(I) = B(I)
 100  IF ( C(I) .GT. 0 ) A(I-1) = C(I)
```

Stores into overlapping sections of the same array must be done in the same order as in the scalar loop.

"Potential multiple store conflict -- use directive if ok"

```
      DO 100 I=1,N
      A(I+J)=B(I)
 100  A(I+K)=C(I)
```

The loop above has a potential overlap between the two stores into A; usually in these situations there is no real overlap between the two sections of A and the NODEPCHK directive should be used to allow the loop to translate.

"Feedback of array elements (equivalenced arrays)"

Actual feedback between arrays equivalenced together.

"Potential feedback (equivalenced arrays) -- use directive if ok"

```
      COMMON /BLOCK/ A(999)
      EQUIVALENCE (S,A(100))
      .
      .
      .
      DO 67 I = M, N
         S = B(I)**2
         A(I) = S + 1.0/S
  67  CONTINUE
```

Either two arrays or an array and a scalar are equivalenced, and their storage relationship in the loop cannot be determined. Use the NODEPCHK or NOEQVCHK directives or the -yp or -ye switches to allow translation, if there is in fact no recursion.

"Equivalence of scalars prevents translation - use directive if ok"

```
      COMMON / BLOCK / A(99)
      EQUIVALENCE (A(1),S), (A(2),T)
      .
      .
      .
      DO 68 J = M, N               (Not translated.)
         S = B(I)**2
         T = C(I)**2
         D(I) = SQRT(S+T)
 68   CONTINUE
```

Two scalars are in the same equivalence class and at least one is modified in the loop. The NODEPCHK or NEQVCHK directives, or the -yp or -ye switches, may be used to allow translation, if in fact there is no recursion.

"Too many data dependency problems"

The loop has exceeded the maximum number of data dependency conflicts allowed (10). Translation of the loop is abandoned at this point to avoid printing out further messages.

Translation Diagnostics

Translation diagnostics point out constructs that prevent a loop from being translated.

Statement types

These messages complain about types of statements that prevent translation.

"ASSIGN prevents loop translation"

```
      DO 210 I = 1, N
         IF ( A(I) .GT. 0 ) ASSIGN 225 TO TOGO
 210  CONTINUE
```

"ASSIGNed GOTO prevents loop translation"

```
      DO 220 I = 1, N
         GOTO TOGO
 220  CONTINUE
 225  CONTINUE
```

"Computed GOTO prevents loop translation"

```
      DO 230 I = 1, N
         GO TO ( 231, 232, 233 ) IGO(I)
 231     B(I) = 0
 232     B(I) = B(I) + A(I)
 233     B(I) = B(I) * C(I)
 230  CONTINUE
```

"I/O statements prevent loop translation"

```
      DO 240 I = 1, N
         WRITE ( IOUNIT ) A(I)*B(I)+C(I)
 240  CONTINUE
```

"RETURN prevents loop translation"

```
      DO 250 I = 1, N
         IF ( X(I) .LT. 0 ) RETURN
         Y(I) = SQRT(X(I))
 250  CONTINUE
```

"STOP prevents loop translation"

```
      DO 260 I = 1, N
         IF ( X(I) .LT. 0 ) STOP 99
         Y(I) = ALOG ( X(I) )
 260  CONTINUE
```

Branches

The messages below relate to handling of conditional operations.

"Backward transfers prevent loop translation"

```
      DO 107 I=1,N
 104  A(J) = SQRT(B(J-1) + D(I))
      J = J + 1
      IF (B(J) .GT. LIMIT) GO TO 104
 107  C(I) = A(J-1)
```

The branch to label 104 is backward, not forward, and prevents translation. In some cases however, backward transfers may be converted into separate DO loops.

"Branches out of the loop prevent translation"

```
     DO 108 I=1,N
     IF (A(I).LT.0) GO TO 777
 108 B(I)=6.0
```

The label "777" is not in the loop.

"Branching too complex to translate this loop"

Because of limits on time and table space, conditional constructs with more than six simultaneously active conditions (i.e. IF-THENs, IF-GOTOs, etc.) cannot be converted to array syntax.

External references

The messages below deal with external references in loops.

"Subroutine call prevents loop translation"

```
      DO 110 I = 1,N
      A(I) = SQRT(B(I)/C(I))
      CALL REVAMP(A(I),D(I))
 110  D(I) = EXP(A(I)+X)
```

"Reference to function that has no array version"

```
     DO 115 I=1,N
 115 A(I) = MYFUNC(B(I))
```

References to non-intrinsic functions in a loop prevent translation of the loop.

DO statement

These messages relate to DO statements.

"DO statement parameters must be integer for array translt."

```
      DO 100 W = 1.0001, 1000000.
         A(I) = W
 100  CONTINUE
```

Non-integer variables or constants are not allowed as the loop index or in the start, end or increment fields for translation purposes to array syntax.

"User function references not allowed in iteration count"

```
      DO 210 I = 1, NLEN(J,K)
 210  A(I)=0.
```

In this example NLEN is an external user function. Such functions cannot appear in the iteration count for a DO loop (they cannot be in the start, end or increment fields of the DO statement) for the loop to be translated to array syntax. Statement functions are allowed, however.

Outer loops

"Outer loop sets inner loop iteration count"

```
      DO 350 J = 1, N
         DO 350 I = 1, J
            S = S + D(J,I)
 350  CONTINUE
```

For an outer loop to be translatable, the iteration count of any inner loops must not change between passes of the outer loop. Here the inner loop is translated, but not the outer.

"Outer loop conditionally executes inner loop"

```
      DO 361 J = 1, N
         IF ( A(J) .LE. 0 ) THEN
            DO 360 I = 1, M
               D(J,I) = SQRT(A(J)*D(J,I))
 360        CONTINUE
         ENDIF
 361  CONTINUE
```

Similarly, the inner loop will be translated, but not the outer.

"No indexing done along this loop"

```
      DO 371 MQ = 1, 1000
         DO 370 I = 1, N
            S = S + A(I)
 370     CONTINUE
 371  CONTINUE
```

The outer loop does no useful work, and is not translated.

"Inner dependence creates outer dependence"

```
      DO 381 J = 1, N
         DO 380 I = 1, M
            A(I) = A(I-1) * B(J)
 380     CONTINUE
 381  CONTINUE
```

In some situations, recursion in the inner loop prevents the outer from translating as well.

Miscellaneous

These messages fall into none of the previous categories.

"Null loop body"

```
      DO 111 I=1,N
 111  CONTINUE
```

Nothing in the loop. (The loop is not eliminated.)

"Character data type inhibits loop translation"

```
      DO 200 I = 1, N
         P(I)(1:2) = Q(I)(2:3)
 200  CONTINUE
```

Use of character type data prevents a loop from being considered for translation.

Warnings

Potential Errors

These warning messages relate to potential errors in the input program.

"* Variable used but never defined ***"**

A local variable is used in executable statements but is never defined.

"Variable defined but never used"

A local variable is defined in executable statements but is never used.

"* Variable used but not defined ***"**

A local variable is used when it is undefined, although it is defined elsewhere in the program unit.

"Variable defined but not used"

This definition of a local variable is not used, although the variable is used elsewhere in the program unit.

"Variable appears only in argument list"

A local variable appears only once, in the argument list of a subroutine call.

"Dead code"

Flags a section of code which, because of the program's flow of control, can never be executed.

Obsolescent Features

Additional warnings are generated for obsolescent features that are no longer preferred usage.

Syntax errors

There are a very large number of syntax error messages. These messages are for the most part self explanatory, and so are not repeated here. .

Internal Errors

Please report any VAST/77to90 internal errors immediately to your support representative.

"Internal error detected (*phase*)-- please report"

Directive Errors

These messages describe errors in the way directives to VAST/77to90 have been used.

"Unknown directive -- it is ignored"

```
CVD$  SCALARIZE
```

"Switch input error"

```
CVD$  SWITCH=-1
```

"Excess characters following directive"

```
CVD$  SKIP THIS LOOP, PLEASE
```

In this case, the characters following SKIP are invalid.

Notes

Notes are not generated by default. They can be enabled with the -po switch.

"IF loop converted to DO-loop"

An IF loop has been converted to a DO loop. (The resulting DO loop may or may not be translatable to array syntax.)

Questions or Comments

Your feedback is important to us. If you have any questions, suggestions, or comments about VAST/77to90 or this document, please send them to:

Pacific-Sierra Research Corporation
Computer Products Group
VAST/77to90 Product Manager
2901 28th Street
Santa Monica, CA 90405

Phone: (310)-314-2300

Fax: (310)-314-2323

Email: info@psrv.com

Index